LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Ilgr
no. 391-396
cop. 2

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

OCT 7 1975
SEP 15 RECD

L161—O-1096

IlGv
no.396
Cop 2

Report No. 396

Math.

TWINKLE--A SYNTAX LANGUAGE FOR A TRANSLATOR WRITING SYSTEM

by

Robert Leroy Mercer

ILLIAC IV Document No. 218

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Digitized by the Internet Archive
in 2013

http://archive.org/details/twinklesyntaxlan396merc

Report No. 396

TWINKLE--A SYNTAX LANGUAGE FOR A TRANSLATOR WRITING SYSTEM*

by

Robert Leroy Mercer

May 15, 1970

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. USAF 30(602)-4144 and submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, February 1970.

ABSTRACT

TWINKLE is a language designed to aid in the syntactic specification of programming languages. In addition to the constructs available in BNF, TWINKLE provides for easy specification of lists and other frequently used linguistic structures. By providing a large number of alternatives for its various constructs, TWINKLE allows the language designer to specify a language in terms that approach natural language.

The implementation of a compiler for TWINKLE is described. This compiler is the first phase of the ILLIAC IV Translator Writing System.

ACKNOWLEDGEMENTS

The author wishes to express his appreciation for the advice and efforts of Dr. Robert S. Northcote who has helped immeasurably in the creation of this paper.

Thanks are also due the author's colleagues Alan Beals, Nelson Machado, and Jacques LaFrance whose contributions to the language described herein and discussions on the translator writing system have been invaluable.

For financial support, I wish to acknowledge the National Science Foundation for its award of a fellowship. I also acknowledge support by the ILLIAC IV project for willing provision of the necessary computer and other physical facilities.

Finally, deep gratitude is expressed to Mrs. Sandy McCabe and Mrs. Shirley Brown for their time and effort in typing the manuscript.

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	1
	1.1 Backus Naur Form	2
	1.2 Translatable Backus Naur Form	3
2.	THE TWINKLE METALANGUAGE FOR SYNTACTIC SPECIFICATION	4
	2.1 Syntactic Symbols	7
	2.1.1 Terminals	7
	2.1.1.1 Characters	7
	2.1.1.2 Special Words	9
	2.1.1.3 Character Mode Terminals	10
	2.1.1.4 Meta-Terminals	11
	2.1.1.5 Blanks	11
	2.1.2 Nonterminals	12
	2.1.3 Any Symbols	12
	2.1.4 Square Brackets	14
	2.1.5 Maybe Symbols	17
	2.1.6 Enclosures	17
	2.1.7 Unordered List	19
	2.1.8 Precedence Structures	19
	2.1.9 Lists	20
	2.1.9.1 Head	20
	2.1.9.2 Type	22
	2.1.9.3 Base	22
	2.1.9.4 Separator	22
	2.1.9.5 Tail	23
	2.1.10 Seeded Lists	24
	2.2 Semantic Symbols	25
	2.2.1 Actions and Tests	25
	2.2.2 Simple Calls	26
	2.2.3 Declaration Calls	27
	2.2.4 Implicit Calls	27
	2.2.5 Parameterized Semantic Calls	28
	2.2.6 Bit Actions and Tests	28
	2.2.7 The Tail	29
	2.2.8 Placement of Calls	29
	2.3 Null and Empty Symbols	31
3.	CONTROL OF THE TWINKLE TRANSLATION	32
	3.1 The Translator Writing System	32
	3.2 Control Statements	35
	3.2.1 Language Name Designation	35
	3.2.2 Print Options	35
	3.2.3 The Parser Type Option	38
	3.2.4 Zip Control	38
	3.2.5 Program Parameter Control	39

Page

3.2.6 Executable Compiler Options 39
3.2.7 Miscellaneous Control Options 40
3.3 Burroughs B-5500 Control Cards for Executing TWINKLE 41

4. IMPLEMENTATION OF THE TWINKLE TRANSLATOR 43

4.1 NONTAB, SYMTAB, and OPRTAB 43
4.2 PROTAB, PRODS and PDLIST 45
4.3 PRODSTACK and LPSTACK 49
4.4 Any Patterns 53
4.5 Grammar Transformations 53
4.5.1 Back Context Absorption 53
4.5.2 Empty Removal 57
4.5.3 Back Substitution of Singly Defined Nonterminals 58
4.5.4 Dummy Insertion 59
4.6 TWINKLE Output Files 61
4.6.1 TABLESF 61
4.6.2 ACTIONS 64
4.7 ZIP Files 65

5. SUMMARY 66

APPENDIX

A. Reserved Words for TWINKLE 68
B. The Syntax of TWINKLE written in TWINKLE and in BNF 69

LIST OF REFERENCES 93

LIST OF FIGURES

Figure		Page
1.	The syntax of the list with separator	21
2.	Block diagram of the Translator Writing System	33
3.	An entry in the NONTAB table	44
4.	PROTAB word format	46
5.	The format of a PRODSTACK	49
6.	The format of a LPSTACK	50
7.	Left recursive and right recursive lists	52

1. INTRODUCTION

The recent proliferation of digital computers has spawned an ever increasing number of formal languages for computer programming and related purposes. Creating a compiler for such a formal language is a decidedly non-trivial task, often requiring several man-years of effort. Therefore, from this bourgeoning stock of languages and compilers, several widely applicable compiler writing techniques have been extracted which at once lead to a deeper understanding of the compiler writing process and to a considerable reduction of the effort involved.

Because of its importance in obtaining a clear and precise definition of a formal language, the development of syntax metalanguages has been intimately related to the development of compiler writing techniques. These metalanguages range from Backus Naur Form (BNF) [1], and its many variants [2, 3, 4, 5], to languages more suitable to syntactic recognition, such as the Floyd production language (FPL) [6] and operator precedence tables [7], and even to the conventional programming languages, FORTRAN [8] and PL1 [9]. Each of these languages has certain advantages: relative compactness and clarity of syntactic structure in the case of BNF and its derivatives; a very clear and explicit statement of the recognition algorithm in the case of FPL and operator precedence tables; and, finally, virtually immediate implementation in the case of FORTRAN and PL1. As is to be expected, ease of producing a linguistic description decreases rapidly as the description itself approaches an implemented compiler.

The primary aim of the TWINKLE metalanguage is to provide a major increase in the ease with which a syntactic specification may be created by a language designer and in the ease with which that syntax may be understood by a user of the language unfamiliar with metalanguages in general. This has

been achieved through the introduction of a wide variety of syntactic symbols for designating many of the common syntactic structures such as lists, enclosures, etc., and through the provision of numerous English words and phrases, which may be used with commonly understood meanings, as an integral part of the syntactic specification.

As a secondary aim, TWINKLE has been designed to present a unified front for the University of Illinois Translator Writing System (TWS). TWINKLE is the input language and the TWINKLE translator is the first phase of the TWS. Thus, TWINKLE combines BNF as described by Beals [3] and translatable BNF (TBNF) as described by Trout [4]. Before progressing to a detailed description of the TWINKLE language and translator which occupies the remainder of this thesis, a brief description is provided of both BNF and TBNF.

1.1 Backus Naur Form

The basic unit of the BNF description of a language is the production. A production consists of a nonterminal (the left hand side) followed by the symbol, triple "::=", followed by a list of terminals and nonterminals (the right hand side). Each nonterminal consists of a string of characters enclosed in either quotes ("__") or angle brackets (<——>). The string may not include ", <, or > and may not start with *. Terminals are special words (strings of alphanumeric characters preceded by #) or characters (A, B, C, etc.). Characters used in the metalanguage (#, ", <, /, etc.) must also be preceded by a # when used as terminals. Two productions with the same left hand side may be combined into one by including the right hand side of the second with the right side of the first and separating it from the latter with the metacharacter, "/". Productions themselves are separated from one another by the metacharacter, ";".

1.2 Translatable Backus Naur Form

TBNF is, in itself, a large step toward simplifying syntactic specification. In addition to the BNF structures described in the last section, TBNF allows:

(i) Kleene star--

<A> * ≡ λ | <A> | <A> <A> | ... ;

(ii) Ampersand for optional presence of some symbol--

<A> & ≡ <A> | λ ;

(iii) Square bracket construct to delimit groups of symbols and alternatives--

<X> ::= <Y> [<A> | <B>] Z

is equivalent to

<X> ::= <Y> <D> Z

<D> ::= <A> | <B> ;

(iv) list <A> ≡ <A> <A> * ;

(v) list <A> separator <B> ≡ <A> [<B> <A>] * ;

(vi) <ANY> ≡ Any symbol at all ;

(vii) but <A> used in conjunction with <ANY> to reduce its generality.

Thus TBNF is considerably more general than BNF. Note, however, that TBNF does not allow left recursion because the parser generated employs recursive descent.

2. THE TWINKLE METALANGUAGE FOR SYNTACTIC SPECIFICATION

When work was begun on the TWINKLE language, two metalanguages, BNF and TBNF, were already in use at the University of Illinois as input languages to the TWS. The BNF input yielded a deterministic parsing algorithm based on the Floyd Production Language (FPL), as described by Beals [3]; while TBNF input yielded a recursive descent (RD) parsing algorithm, as described by Trout [4]. Each parser has certain advantages but, once either BNF or TBNF has been chosen as the metalanguage, it requires a major effort to convert the description to the alternate form. Thus, although it would be desirable, because of its relatively rapid generation, to create a RD parser during the debugging phases of the language description, it would be equally desirable, because of the rigorous exclusion of ambiguity inherent in the nature of the deterministic FPL parser, to create a FPL parser when the last phase of the compiler creation is reached. Unfortunately this ideal has not been attainable in the past, primarily due to the difficulty of translating TBNF into BNF by hand. These considerations, therefore, dictate that TWINKLE be a superset of both BNF and TBNF, so that existing language specifications may be accepted by the new system with little or no change, and that the TWINKLE translator output be either BNF or TBNF.

The basic form of TWINKLE, therefore, is cast in the familiar BNF mode. That is, a TWINKLE syntactic specification consists of one or more productions, each of which has a left hand side, which is the nonterminal being defined (wholly, or in part) by the production, and a right hand side which comprises a set of alternative definitions for the

nonterminal in the left hand side. Each definition, in turn, comprises a string of TWINKLE syntactic and semantic symbols. In BNF the productions are separated from one another by semicolons; the definitions by vertical bars which are actually rendered in the implementation by a slash; and the left and right hand sides of a production by the character, triple "::=" . It is one of the aims of the TWINKLE project to make possible the rigorous specification of a language syntax in a way that is at once acceptable to both human beings and computers. To this end, English phrases have been provided for the replacement and/or embellishment of the metalinguistic symbolism of BNF and TBNF. In addition, several features not present in either BNF or TBNF have been introduced. For example, the BNF productions,

```
<PROGRAM> ::= #BEGIN <LIST OF STATEMENTS> #END /
                    #BEGIN #END;
<LIST OF STATEMENTS> ::= <STATEMENT> / <LIST OF STATEMENTS> #;
                <STATEMENT>;
```

may be written in TWINKLE in the much more readable form:

```
A <PROGRAM> CONSISTS OF A POSSIBLY EMPTY LIST OF <STATEMENT>S
SEPARATED BY SEMICOLONS ENCLOSED IN #BEGIN AND #END;
```
(1)

At first glance this might appear to have two distinct interpretations: particularly, a program may be either

```
BEGIN s ; s ; s ; ... ; s END
```

or

```
s BEGIN; END s BEGIN; END s ... BEGIN; END s
```

where "s" here stands for <STATEMENT>. In fact, the former interpretation is made. If the language designer wishes to express the latter form of program, he may write:

A <PROGRAM> CONSISTS OF A POSSIBLY EMPTY LIST OF <STATEMENT>S
SEPARATED BY [SEMICOLONS ENCLOSED IN #BEGIN AND #END]; (2)

In TWINKLE, the square brackets ([]) serve the function of delineating clause and phrase structure in productions. The enclosure operator always acts on the immediately preceding syntactic symbol which is at the same nesting level as the enclosure operator, itself. Thus, in production (1), it is the possibly empty list that is to be enclosed and not the semicolons. In production (2), on the other hand, the square brackets associate the enclosure operator with the semicolons and indicate that this construct, taken as a whole, is to be the separator of the possibly empty list.

It is to be noted, however, that the TWINKLE language does not enforce strict grammatical usage of English but, rather, allows for such usage by the language designer. Thus, it will be found that, in the TWINKLE syntax, the articles "A" and "AN" are treated equally with the result that a determined corruptor of English might write (1) as

AN <PROGRAM> CONSISTS OF AN POSSIBLY EMPTY LIST OF <STATEMENT>S
SEPARATED BY SEMICOLONS ENCLOSED IN #BEGIN AND #END; (3)

The TWINKLE program, in its most general form, is made up of three distinct portions, of which the language syntactic description is the second. The first portion is a set of control statements which determine, among other things, the nature and volume of the TWINKLE output and

the processing options for the remainder of the TWS. The construction and use of this portion is dealt with in Chapter 3. The third portion, the semantic tail, conveys the necessary semantic information about the language being described. This is discussed briefly in 2.2.5, and more fully by Machado [10]. The remainder of the current chapter is a discussion of the syntactic and semantic symbols available to the language designer for the syntactic portion of the TWINKLE system.

2.1 Syntactic Symbols

2.1.1 Terminals

Terminals are those symbols of which the object program is ultimately composed. They may occur only on the right hand side of a production and are represented in the syntax in a number of different ways. Terminals fall into the following classes: characters, special words, character mode terminals, meta- or special class terminals, and blanks.

2.1.1.1 Characters

The simplest form of the terminal is the character. Here character refers to any of the twenty-seven special characters accepted by equipment of the Burrough's B-5500 computer system. These are included, together with the other thirty-seven B-5500 characters, in Table I. In the syntax, a character may be represented by prefixing it with a sharp (#) when the character standing alone would have some special significance to TWINKLE (i.e., when the character is a meta-character). For example, a comma is represented in the syntax by the symbol pair "#,". However, since more than half of the special characters available are meta-characters, it is probably safest to include the sharp in all cases. This is a point at which TWINKLE diverges from the standard BNF and TBNF. The latter have

Table I. The Burroughs B-5500 Character Set
Classes of Any Base Symbols

Terminal	Internal Code	Any Bases*	Terminal	Internal Code	Any Bases*	Terminal	Internal Code	Any Bases*	Terminal	Internal Code	Any Bases*
0	0	0,1,3	A	17	0,1,2	K	34	0,1,2	T	51	0,1,2
1	1	0,1,3	B	18	0,1,2	L	35	0,1,2	U	52	0,1,2
2	2	0,1,3	C	19	0,1,2	M	36	0,1,2	V	53	0,1,2
3	3	0,1,3	D	20	0,1,2	N	37	0,1,2	W	54	0,1,2
4	4	0,1,3	E	21	0,1,2	O	38	0,1,2	X	55	0,1,2
5	5	0,1,3	F	22	0,1,2	P	39	0,1,2	Y	56	0,1,2
6	6	0,1,3	G	23	0,1,2	Q	40	0,1,2	Z	57	0,1,2
7	7	0,1,3	H	24	0,1,2	R	41	0,1,2	,	58	0,1,4
8	8	0,1,3	I	25	0,1,2	S	42	0,1,4	%	59	0,1,4
9	9	0,1,3	.	26	0,1,4	*	43	0,1,4,6	≠	60	0,1,4,5
#	10	0,1,4	[	27	0,1,4	-	44	0,1,4,6	=	61	0,1,4,5
@	11	0,1,4	&	28	0,1,4	)	45	0,1,4	]	62	0,1,4
?	12	0,1,4	(	29	0,1,4	;	46	0,1,4	"	63	0,1,4
:	13	0,1,4	<	30	0,1,4,5	≤	47	0,1,4,5	special		word-0,7
>	14	0,1,4,5	←	31	0,1,4		48	0,1	<*I> -		0,7
≥	15	0,1,4,5	x	32	0,1,4,6	/	49	0,1,4,6	<*N> -		0,7
+	16	0,1,4,6	J	33	0,1,2	S	50	0,1,2	<*S> -		0,7

* 0 = Any Terminal
1 = Any Character
2 = Any Letter
3 = Any Digit
4 = Any Special Character
5 = Any Relational Operator
6 = Any Algebraic Operator
7 = Any Non-Character

fewer meta-characters and, as such, require fewer sharps. Although it is very easy to insert the necessary sharps, it may be desirable to make a preliminary run through the TWINKLE translator alone to isolate what trouble spots there may be.

An alternative method for indicating a character in the syntax which avoids the details of the sharp, and which provides for a more readable syntax, consists of writing down the English word or phrase which identifies the character in question. Thus, in production (1) above, SEMICOLONS is used in place of the equally acceptable "#;". While this form is not available for all of the special characters, it proves quite useful in practice. A complete list of these alternatives is given in the TWINKLE syntax (see Appendix B).

2.1.1.2 Special Words

Many times it is convenient to consider a group of letters and digits as being, conceptually, a single terminal. Thus, in languages of the ALGOL family, the letter strings BEGIN and END are each taken as single terminals. This approach has an advantage in the milieu of the TWS in that these conceptual units, or special words, are compiled relatively quickly by the scanner as opposed to the more laborious and time consuming letter by letter compilation through the syntax and semantics.

Any string of letters and digits which begins with a letter may be used as a special word. It must not have embedded blanks, and the character immediately after it must not be alphanumeric. As with characters, a special word must be prefixed by a sharp in the syntax if it would otherwise be of special significance to the TWINKLE translator. Since there are well over one hundred such words in TWINKLE (see Appendix A),

it is safest to use sharps liberally. Again, there is a divergence of TWINKLE at this point from BNF and TBNF which is easily overcome.

2.1.1.3 Character Mode Terminals

While the special word is often the better way of entering alphanumeric information, there are times when character by character input is actually preferable. For example, the parsing of FORTRAN and B-5500 ALGOL FORMAT statements is simplified if done in the character mode. More generally, any time there is an abundance of single character significance in a syntactic entity, it is better parsed and more compactly described in the character mode.

If the sequence of characters to be dealt with consists entirely of digits, it may be written into the syntax directly because an unadorned number has no special significance to the TWINKLE translator. If, however, a more general sequence must be handled, the sequence must be preceded by the word ALPHA, which indicates to TWINKLE that it must consider the following sequence of characters specially. Since ALPHA is a bit long, it behooves one to provide a means of keeping its use to a minimum. To this end, the construct,

[ALPHA A / ALPHA B / ALPHA C / ... / ALPHA Z] ,

is equivalent to the more compact form,

ALPHA [A / B / C / ... / Z] .

Another, more obscure, method of specifying alphanumeric characters (or, in fact, any of the Burroughs B-5500 characters) is the code construct which is based on the internal binary representations of the various characters. This form of character representation is a

carry over from TBNF where it was adopted primarily because the question mark is not a valid character on punched cards in the B-5500 system. It consists of the word CODE followed by an integer between zero and sixty-three which is the internal code of the character being indicated.

2.1.1.4 Meta-Terminals

Because of the advantages attendant to allowing the scanner to perform a certain amount of simple syntactic analysis immediately on the input string (as, for example, in the recognition of special words), the TWS scanner also recognizes members of three special classes of terminals: identifiers, numbers, and strings. These meta-terminals are represented in the syntax by the symbols <*I>, <*N>, and <*S>, respectively. In an English-like syntax, they may be represented by IDENTIFIER (or IDENTIFIERS), NUMBER (or NUMBERS), and STRING (or STRINGS). An identifier is any sequence of alphanumeric characters beginning with a letter, provided the sequence is not a special word of the language. A string is any sequence of characters (excluding the quote) enclosed in quotes. A discussion of how the scanner handles these items has been given by Machado [10].

In addition to these three meta-terminals, TWINKLE allows for the syntactic specification of up to twenty other meta-terminal symbol classes. In the syntax, these are represented by the symbol <*n> where n is a digit between four and twenty-three and identifies the meta-terminal. A special scanner is necessary to take advantage of this facility.

2.1.1.5 Blanks

Blanks are specified in the syntax by the word BLANK. A blank can only be scanned in the character mode.

2.1.2 Nonterminals

Nonterminals are specified in TWINKLE as strings of characters, called nonterminal names, enclosed in either angle brackets or quotes (< ____ > or " ____ "). For obvious reasons the nonterminal name may not include either angle brackets or quotes. Furthermore, any blanks which appear in the nonterminal name are disregarded. Thus, the nonterminals, <PLOUGHS HARE> and <PLOUGH SHARE>, are treated identically. In the BNF or TBNF output resulting from a TWINKLE translation, the blanks in nonterminal names displayed are, in fact, removed. To retain a modicum of readability in this compact form it is advisable to hyphenate multi-word nonterminal names; for example, use <TYPED-PROCEDURE-DECLARATION> in place of <TYPED PROCEDURE DECLARATION>.

The first nonterminal to appear in the syntax (i.e., the nonterminal on the left hand side of the first production), is the unique objective symbol or program symbol of the language. Any string of terminal symbols which may be reduced to this nonterminal is a program in the language.

In addition to the nonterminals which are explicitly defined by the language designer, nonterminals are introduced by TWINKLE in order to implement lists and to execute some of the grammar transformations discussed in Chapter 4. These nonterminals are distinguished by the fact that they all have one or more blanks in their names after translation.

2.1.3 Any Symbols

In BNF, alternatives must be written out explicitly. Thus, to specify a nonterminal, say <LETTER>, which is any of the twenty-six letters of the alphabet, one must write--

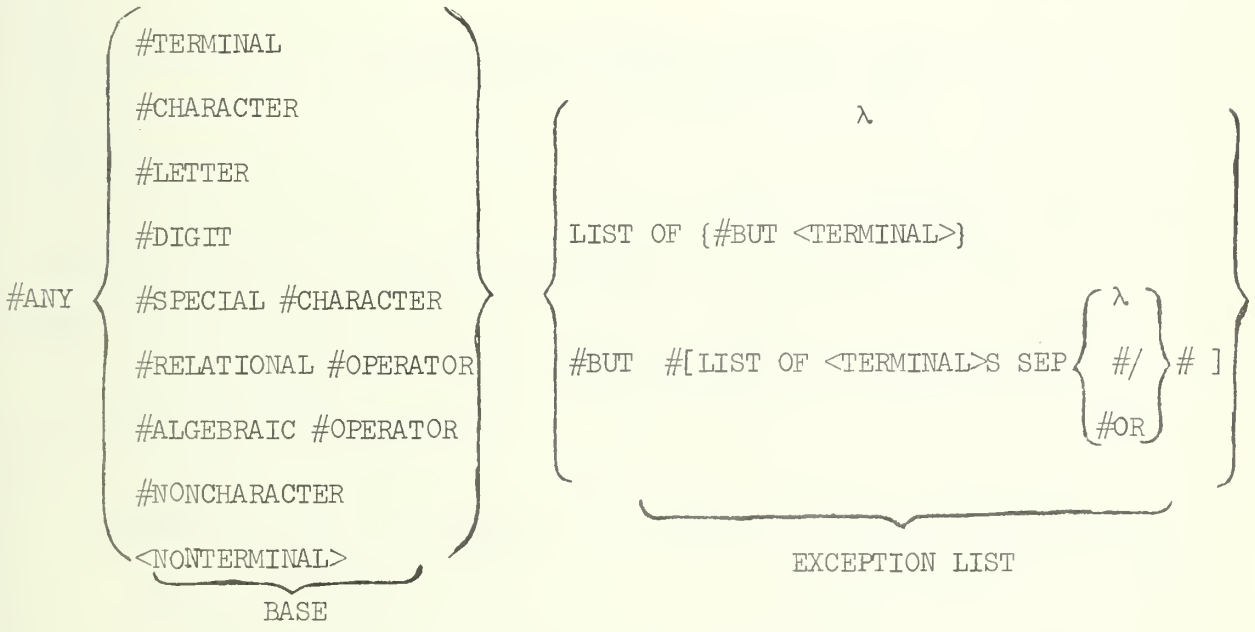
<LETTER> ::= A / B/ C/ ... / X / Y / Z .

Trout, in TBNF, introduced the pseudo-nonterminal, <ANY>, which stands for any terminal symbol. If not all terminals are to be indicated, the exceptions, if small in number, may follow the pseudo-nonterminal--each preceded by the special word, BUT. For example, any terminals except BEGIN and END may be written:

<ANY> BUT #BEGIN BUT #END .

This construct has been used primarily in error recovery in TBNF languages.

In TWINKLE, the any symbol has been generalized and has become a powerful programming tool. The syntax of <ANY SYMBOL> is shown below:



Use has been made of a rather simple two-dimensional extension of TWINKLE: square brackets have been replaced by vertical braces with the alternatives occupying one line each; the Greek letter "λ" is used instead of the special word, LAMBDA.

The terminals in each of the bases, except for <NONTERMINAL>, are shown in Table I (page 8). The <NONTERMINAL> base is unique in that, first,

the elements that it contains depend upon the actual nonterminal symbol used and, second, they are not restricted to terminals but include all of the alternative definitions of the nonterminal. Any terminals which are in the base, but which are not desired, may be written in the exception list following the <ANY SYMBOL>. The TBNF form of the exception list is still accepted by TWINKLE but, as with the ALPHA list, it is possible to use only one BUT and enclose the terminals of the exception list in square brackets immediately following it. Thus, in place of

BUT #BEGIN BUT #END BUT #LEFT BUT #RIGHT ,

one may write

BUT [#BEGIN / #END / #LEFT / #RIGHT] .

2.1.4 Square Brackets

In English, clauses are separated at one level by commas and at a higher level by semicolons. Beyond this either more than one sentence is used or the clause separation must be done by the reader from context. Even this, however, does not prevent ambiguity beyond four levels or so. A language such as TWINKLE, in which it is necessary to indicate clause nesting to an arbitrary level, must have a more powerful mechanism available.

Since the semicomma and the demisemicolon do not yet exist, it was decided that clauses and other such ensembles, which are intended as single syntactic entities, be enclosed in square brackets. Two examples of this have already been encountered: the terminal list following ALPHA, and the terminal list following BUT. Beyond these the square brackets find

several other uses; whenever one or more symbols, or groups of symbols, appear at one spot in a production they may be enclosed in square brackets. Thus, the productions,

AN <ANY SYMBOL> CONSISTS OF #ANY #TERMINAL;

AN <ANY SYMBOL> CONSISTS OF #ANY #CHARACTER;

AN <ANY SYMBOL> CONSISTS OF #ANY #LETTER ,

may be written more compactly as

AN <ANY SYMBOL> CONSISTS OF #ANY FOLLOWED BY [#TERMINAL OR #CHARACTER OR #LETTER].

For purposes of adding semantic symbols, which will be discussed later, the special word, ANY, and the bracket construct, taken as a whole, are considered to be at level zero of the production, while the special words; TERMINAL, CHARACTER, and LETTER, are considered to be at level one.

Alternatively, an entire production may be nested in square brackets and one may write--

AN <ANY SYMBOL> CONSISTS OF #ANY FOLLOWED BY AN [<ANY BASE> WHICH IS DEFINED TO BE #TERMINAL OR #CHARACTER OR #LETTER] . (4)

Note that although all previous forms of arrow are still valid in the nested production, it is also permissible to include the special word, WHICH, so that the construct will look more like an English clause. When a nonterminal, such as <ANY BASE>, is defined in a nested production it

may then appear anywhere else in the syntax just as if it had been defined in the usual manner. There are, however, some precautions to be taken with nested productions. These stem from the fact that the nonterminal so defined may have additional definitions elsewhere in the syntax. In this case the nonterminal represents the totality of its alternative definitions, except when it appears on the left hand side of a nested production in which case it represents only those definitions which appear on the right hand side of the same nested production. To illustrate this, suppose that in addition to the production above, in which <ANY BASE> is defined, one also has the production:

A <SOME SYMBOL> CONSISTS OF #SOME FOLLOWED BY AN
[<ANY BASE> WHICH IS #DIGIT OR #DIGITS] . (5)

The two productions (4) and (5) are then equivalent to the BNF productions:

<ANY SYMBOL> ::= #ANY <ANY BASE 1> ;

<SOME SYMBOL> ::= #SOME <ANY BASE 2> ;

<ANY BASE> ::= <ANY BASE 1> / <ANY BASE 2> ;

<ANY BASE 1> ::= #TERMINAL / #LETTER / #CHARACTER ;

<ANY BASE 2> ::= # DIGIT / #DIGITS .

The square bracket construct may also be applied to a single string of symbols to indicate that the string is to be taken as a unit itself. This is useful in constructs such as the maybe symbol, enclosures, and lists described below.

2.1.5 Maybe Symbols

Frequently a syntactic structure has one or more substructures which may be omitted without syntactic error. Thus, for example, in ALGOL a list of labels preceding a statement is optional. Many other examples may be found in algorithmic languages; several may be found in TWINKLE, itself. To make the specification of such structures as easy as possible, Trout [4] adopted the Brooker and Morris question mark [11]--changing it, in the process, into an ampersand because the question mark is an illegal character on B-5500 cards. In TWINKLE, either an ampersand or a question mark may be placed after a symbol (or group of symbols enclosed in square brackets) to indicate that it is optional. The English-like form of this construct consists of preceding the optional symbol by the special word phrase, POSSIBLY ONE. This form is more general, in that it may be applied directly to lists and enclosures; whereas, they must be enclosed in brackets when followed by a question mark or ampersand.

An example of the English form of <MAYBE SYMBOL> follows. The production,

AN <ANY SYMBOL> CONSISTS OF AN <ANY PART> FOLLOWED BY

POSSIBLY ONE <EXCEPTION LIST> ,

is equivalent to the two BNF productions:

<ANY SYMBOL> ::= <ANY PART> ;

<ANY SYMBOL> ::= <ANY PART> <EXCEPTION LIST> .

Under more complex conditions the maybe symbol can account for a considerable increase in readability of the syntax.

2.1.6 Enclosures

Another very common construct in computer languages is that in which some structure, such as a list of subscripts, is enclosed in delimiters, such

as parentheses. The delimiters may be different: e.g., the special words, BEGIN and END, which bracket compound statements in ALGOL; they may be different but very closely related: e.g., the left and right parentheses which enclose subscript lists in FORTRAN and PL1; they may be identical: e.g., quotes which delimit strings in ALGOL or periods which enclose logical operators in some dialects of FORTRAN. Corresponding to these possibilities there are three forms of enclosure. The general representative of the first form may be symbolized as:

<SYMBOL> #ENCLOSED #IN <SYMBOL> #AND <SYMBOL>

where <SYMBOL> represents any basic TWINKLE symbol or group of symbols enclosed in brackets. The latter two symbols, if not enclosed in brackets, may not be enclosure symbols themselves. Using this construct and the list symbol discussed below a compound statement in ALGOL may be defined by the very clear production:

A <COMPOUND STATEMENT> IS DEFINED TO BE A POSSIBLY EMPTY LIST OF <STATEMENT>S SEPARATED BY SEMICOLONS ENCLOSED IN #BEGIN AND #END; .

The second form of enclosure actually applies to only three sets of characters in the Burroughs character set: parentheses, square brackets, and angle brackets. It may be symbolized, using the two dimensional bracket construct described earlier, as:

<SYMBOL> #ENCLOSED #IN { #PARENTHESES / #ANGLE #BRACKETS / #SQUARE #BRACKETS } .

Finally, the third form of enclosure symbol is simply:

<SYMBOL> #ENCLOSED #IN <SYMBOL>

where special word, S, may be added to make the second symbol plural.

2.1.7 Unordered List

The unordered list provides a means of indicating that a group of items may appear in any order. One very simple example of the use of this is the PL1 iterated DO loop. The leading statement may have an initial value, increment, and final value for the control variable; the increment and final value may appear in either order. Symbolically the unordered list has the form:

#UNORDERED #LIST #OF [<SYMBOL>#AND<SYMBOL>... #AND <SYMBOL>] ;

or

[<SYMBOL> #AND <SYMBOL> ... #AND <SYMBOL>] #IN #ANY #ORDER .

2.1.8 Precedence Structures

The precedence structure was introduced in TBNF to allow for the specification of operator precedence in a BNF environment. Since the precedence structure does not lend itself readily to a simple English alternative, its syntax has not been expanded from the TBNF version. A precedence structure consists of: the special word, OPERATOR, followed by a list of precedence groups enclosed in square brackets; followed by the special word, ON; and, finally, by the operand on which the precedence is based. Each precedence group consists of a list of symbols, the operators, followed by the special word, PRECEDENCE, and a pair of integers separated by a comma enclosed in parentheses. The integers indicate the precedence of the preceding operators in the stack and in the input stream, respectively. Succeeding precedence groups are separated from one another by slashes. The following is an example of the use of a precedence structure:

<ARITHMETIC EXPRESSION> ::=

OPERATOR[#+ #- PRECEDENCE(1,1)/ #/ #× PRECEDENCE(2,2)] ON <PRIMARY> .

2.1.9 Lists

The list is one of the most useful of the TWINKLE constructs. Consequently it occurs in a wide variety of English-like and symbolic forms. To indicate a simple list without separators, the right square bracket of a nested production or square bracket construct may be followed by an asterisk or by a plus sign. The asterisk denotes a possibly empty list while the plus sign denotes a list that must have at least one element.

In BNF, lists are usually formed either by left recursive productions such as:

<A> ::= <B> / <A> <B> ;

or by right recursive productions such as:

<A> ::= <B> / <B> <A> .

This underlying structure is masked by the simplicity of the TWINKLE constructs but may be made explicit, if desired, by the insertion of the qualifying special word, OPEN, for left recursion or CLOSE for right recursion between the right square bracket and the asterisk. If no qualifier is present, left recursion is implied.

The syntax of list structures allowing for separators is shown in figure 1. As indicated it consists of five portions: head, type, base, separator, and tail. Only the type and base are, necessarily, nonempty but at least one of the head and tail must be empty. The functions of these various components are discussed below.

2.1.9.1 Head

In the absence of the list tail, the list head determines whether the list is left or right recursive. If it is empty (the special word, REDUCED, or the phrase, LEFT RECURSIVE), the list is left recursive. If it is R, EXPANDED, or RIGHT RECURSIVE, the list is right recursive.

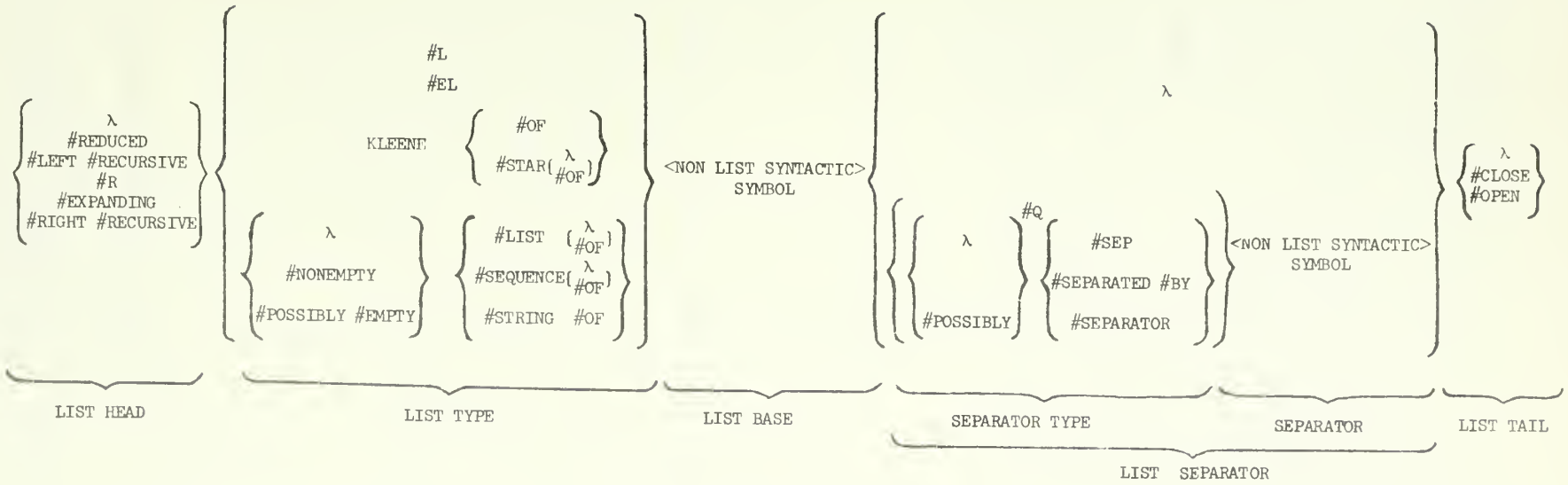


Figure 1. The syntax of the list with separator.

2.1.9.2 Type

Lists may be either possibly empty or nonempty, as noted above. It is the function of the list type, which may not be empty, to determine this characteristic. A nonempty list is indicated by the list types: L, LIST, STRING, SEQUENCE, NONEMPTY STRING, NONEMPTY LIST, and NONEMPTY SEQUENCE. For a possibly empty list, the available types are: EL, KLEENE, KLEENE STAR, POSSIBLY EMPTY LIST, POSSIBLY EMPTY STRING, and POSSIBLY EMPTY SEQUENCE. To improve readability, the list type may be followed by the special word, OF. If the list type is either STRING or KLEENE, OF is necessary to avoid syntactic ambiguity.

2.1.9.3 Base

The base may be a single symbol or group of symbols enclosed in square brackets. It must not be a list itself and is considered to be nested one level deeper than the list of which it is the base. An S may follow the base, in some cases, to indicate its plural character in the grammatical structure of the list. In addition, phrases which may be used as character terminals all have plural forms which may be used profitably here.

2.1.9.4 Separator

The power of the list construct is greatly enhanced by the possibility of specifying a separator. Like the base, the separator may be either a single symbol or a group of symbols enclosed in square brackets. The separator, like the base, is considered to be at the next

higher bracket nesting level from that of the list, itself. The plural forms valid in the base are also valid in the separator. The appearance of the separator between successive base items is either required or optional--according as the separator type is definite or questionable. Definite separator types are indicated by SEP, SEPARATOR, and SEPARATED BY; questionable separator types are indicated by Q, and by the special word, POSSIBLY, followed by any one of the definite separator types.

2.1.9.5 Tail

In the absence of a list head, the tail determines whether the list takes the left recursive or right recursive form. If it is empty, or the special word, CLOSE, the list is left recursive. If it is the special word, OPEN, the list is right recursive. Note once again that either the tail or the head must be empty in any list structure.

The examples below illustrate the various aspects of the list structure:

```
A <SYNTACTIC DESCRIPTION> CONSISTS OF A POSSIBLY
        EMPTY LIST OF <PRODUCTION>S SEPARATED
        BY SEMICOLONS;

<ARITHMETIC EXPRESSION> ::= LIST [LIST <PRIMARY>
        SEP [#* /#/ ]] SEP [#+ / #-];
```

<STATEMENT> ::= [<LABEL> #:] * <UNLABELED STATEMENT>;

AN <ANY BASE> CONSISTS OF A LEFT RECURSIVE LIST OF
<TERMINAL>S POSSIBLY SEPARATED BY SLASHES
ENCLOSED IN SQUARE BRACKETS; .

2.1.10 Seeded Lists

Any of the list structures, the syntax for which appears in figure 1 above, becomes a seeded list when followed by a list seed. The syntax of the list seed is

$$\left\{\begin{matrix}\#\text{STARTING}\\ \#\text{BEGINNING}\end{matrix}\right\}\ \#\text{WITH}\ \langle\text{SYMBOL}\rangle\ ,$$

where <SYMBOL> may be any TWINKLE symbol, or group of symbols, enclosed in square brackets, with the exception of enclosures and maybe symbols.

The seed list may be used to indicate that the first element of a particular list is distinguished in some way from the rest. For example, the syntax of an ALGOL block may be written:

A <BLOCK> CONSISTS OF A POSSIBLY EMPTY LIST OF <STATEMENT>S
SEPARATED BY SEMICOLONS STARTING WITH A POSSIBLY EMPTY
LIST OF <DECLARATION>S SEPARATED BY SEMICOLONS ENCLOSED
IN #BEGIN AND #END .

Note that, since the list seed may not be an enclosure, the enclosure operator applies to the entire seeded list and not simply to the list of <DECLARATION>S.

2.2 Semantic Symbols

The TWINKLE symbols and constructs described up to this point may be used in the syntactic specification of a language. A compiler, however, must be more than simply a recognizer for a language. It must assign appropriate meanings (e.g., in the form of equivalent machine code) to the various syntactic entities that it recognizes from the input stream. These meanings, taken as a whole, make up the semantic description of the language or, more simply, the semantics of the language. In the TWS, the semantics is written in the Illinois Semantics Language (ISL), a complete description of which is provided by Machado [10]. For the purposes of this discussion, it suffices to consider the ISL semantic description as a number of individual semantic blocks, each of which is associated with a semantic name through which it may be accessed by the parser. Before describing the manner in which the parser is directed to initiate a semantic block, it is necessary to consider the possible results of the execution of a semantic block.

2.2.1 Actions and Tests

Based on their effect on the parser, semantic blocks may be divided into two groups: semantic actions and semantic tests. Actions have no direct effect on the parser and are used primarily for such functions as table creation, code emission, etc. Tests, on the other hand, are actually more of a syntactic character than a strictly semantic character. Thus, a test is called by the parser when the nature of a particular entity is syntactically undeterminable and requires investigation on a higher level. For example, in an ALGOL assignment statement a boolean variable must be assigned the value of a boolean expression while an integer or real variable must be assigned the value of an arithmetic expression. Since the declaration in which the type of the variable in question was determined may precede its

usage by an arbitrary amount, and since an arithmetic expression and a boolean expression may, in general, be identical for arbitrarily many symbols, the parser is unable to determine whether an arithmetic or boolean assignment is being made. The question is resolved by calling a test which compares the variable with tables made when the declarations were recognized. The result of this comparison is communicated to the parser which then is able to proceed correctly with the parse.

Communication between test and parser is by means of the globally declared boolean variable, SEMANTICTEST. If the value of SEMANTICTEST after the execution of the test is true, the parser continues along the indicated branch of the parsing tree. Otherwise the parser abandons this branch and must decide among the remaining branches, possibly invoking further tests in the process. It is important, therefore, that each block, which may be called as a test, set SEMANTICTEST at some point in its execution. If this is not done, the parser will determine its future course of action from the previous value of SEMANTICTEST with the attendant possibility of an erroneous parse. Since any block may be called, both as an action and as a test, at different times during the parse, it is permissible for an action to set SEMANTICTEST although the variable is disregarded under these circumstances.

An indication to the parser of the points in the syntactic recognition at which a particular action or test must be invoked is given by placing semantic call symbols at appropriate points in the syntax. These calls may have any of the three forms described below.

2.2.2 Simple Calls

Simple action (test) calls consist of the special symbols, "@S"("@T"), followed by the name of the action (test) being called. Any string of digits or alphanumeric characters beginning with a letter may be used as a name.

A particular semantic block may be called from as many places in the syntax as desired and may be called as a test of one point while being called as an action at another point. Several forms of the simple call linger from earlier versions of the TWS. Thus, in addition to "@S", an action name may be preceded either by "@Q" or "#", and a test name may be preceded by "#" and enclosed in either quotes or angle brackets. When "#" is used in an action call, the action name must be a string of digits. The reason for this is that the call would appear to be a special word if the action name were to begin with a letter.

2.2.3 Declaration Calls

Any of the simple calls (except the "#" <action name> form) may be extended into a declaration call. The action or test name is followed by a colon and then a description of the action or test in ISL code. This description, or declaration, must be enclosed either in square brackets, or in the special words, BEGIN and END. Each name may be declared no more than once in this way, although such a declaration is not necessary. Any name so declared may be used in the syntax in simple calls, both before and after the appearance of the declaration. When the semantic block in question is brief, the overall clarity of the linguistic description may be considerably enhanced by using the declaration call.

2.2.4 Implicit Calls

When a block is of such a nature that a declaration call would be appropriate, and yet is used in only one place, it is clearly not necessary to give a name to that block. Under these circumstances the name and colon in the declaration call may be omitted, thereby creating an implicit call. For each implicit call in the syntax, the TWINKLE translator generates a unique name through which the relevant block of code may be referenced by the ISL translator.

2.2.5 Parameterized Semantic Calls

Semantic calls, with the exception of implicit calls, may be modified by a list of integer constant parameters separated by commas enclosed in parentheses and placed immediately following the action or test name. These constants are used by the parser to set a group of global variables (the array row PARAM) that may be referenced by the semantic routine when it is called. This is frequently very useful when a number of portions of the syntax, which would otherwise require different semantic actions, may be serviced by a single, appropriately parameterized, action. For example, in recognizing written out characters, TWINKLE employs a single semantic action whose parameter is the internal code number of the symbol recognized.

2.2.6 Bit Actions and Tests

Frequently, a semantic action involves nothing more than the setting of a single bit. Similarly, a semantic test is frequently based on the condition of such a bit. Calling a semantic block to perform these manipulations requires a disproportionate amount of overhead and it was, therefore, considered appropriate to introduce special action and test types specifically for performing bit operations. The syntax of the bit action is:

$$\#@ \quad \#S \quad \left\{\begin{matrix}\#SET\\ \#RESET\end{matrix}\right\} \quad \#BIT \ \langle BIT\ NAME\rangle$$

where <BIT NAME> is either a number, identifier, or TWINKLE special word. The designated bit is correspondingly either set or reset. The syntax for the bit test is:

$$\#@ \ \#T \ \#BIT \ \langle BIT\ NAME\rangle \underbrace{\left[\begin{matrix}\lambda\\ \#ON\\ \#OFF\end{matrix}\right]}_{\text{Condition}} \quad \underbrace{\left[\begin{matrix}\lambda\\ \#SET\\ \#RESET\end{matrix}\right]}_{\text{Action}} .$$

The test is true if the designated bit is in the condition specified (i.e., ON or OFF). The default condition is ON. If the test is true the indicated action is taken on the designated bit. Up to 48 different bit names may be used by the language designer. These are assigned by the TWINKLE translator to the 48 bits of the global parser variable, ACTIONBITS.

2.2.7 The Tail

In most languages it will not be desirable to declare each block (implicitly or otherwise) directly in the syntax specification. Also, all but the simplest of semantics will require a number of variables and procedures declared globally to the individual semantic blocks. For the sake of completeness, these global declarations and undeclared blocks may be enclosed between the special words, BEGIN and END, (thus forming the semantic tail) and appended to the syntax specification. This tail will then be passed directly to the ISL translator upon completion of the TWINKLE translation. In this way a language may be completely processed by the TWS from a single complete specification of the language. During the debugging phase of the language development, it is more natural to process the syntax and semantics separately. The details of coordinating the ISL translator with the rest of TWS in an independent run are discussed by Machado [10].

2.2.8 Placement of Calls

A semantic call may appear anywhere in the syntax that a syntactic symbol may appear, except at the beginning of an alternative. Thus, a semantic call may not appear immediately after the arrow of a production, the left square bracket of a square bracket construct, or the separator ("/" or OR) of a list of alternatives. The reason for this is that a semantic call cannot be made by the parser until it has determined exactly what stage the parse has reached. Clearly the parser cannot, in general, determine this at the beginning of an alternative.

Unfortunately, there is a good deal more to placing semantic calls than simply knowing where they will be legal. The ideal time to place them would be after the FPL form (see chapter 3 and the paper by Beals [3]) had been generated. The condition of the stack and the phase of the parse

would then be known explicitly. It is, however, fairly straightforward to place them directly into BNF as was done in earlier versions of the TWS. The problems present in TWINKLE, with respect to call placement, arise chiefly from the complex structures (lists, enclosures, etc.) available and from the large amount of grammar transformation that is inherent in TWINKLE translation. Thus, virtually all of the TWINKLE constructs not present in BNF employ some form of TWINKLE generated nonterminals in their implementation. Because of this, the configuration of the stack at the moment of a semantic call may not be easy to determine. The following guidelines will be helpful in creating and placing semantic calls to achieve a given end:

1. A semantic call is made following the recognition of the symbol or construct at the same bracket nesting level immediately preceding its occurrence in the syntax. For example, if one writes

   <A> ::= LIST <B> SEP <C> @S1 ,

   semantic action "1" will be called after the entire list has been recognized and not after each separator, <C>. The latter effect may be achieved by

   <A> ::= LIST <B> SEP [<C> @S1] .

2. A semantic routine should not reference the stack for symbols at the same nesting level as its call--provided that the call and the symbol are separated by one of the non-BNF TWINKLE constructs. For example, in the TWINKLE production,

   <A> ::= <B> <C> LIST <E> <F> <G> @S A ,

   the semantic action, "A", may reliably reference the nonterminals, <F> and <G>, but may not reference the nonterminals, <B> and <C>, from which its call is separated by the list construct.

3. A semantic routine should not reference symbols which occur at different nesting levels from its call. The only exception to this rule is the case in which a semantic call immediately follows the right square bracket of a square bracket construct. The call is then, in essence, copied onto the end of each of the alternatives within the square brackets.

A detailed example showing the placement of semantic calls in a TWINKLE grammar for a subset of ALGOL is provided by Machado [10].

2.3 Null and Empty Symbols

Several forms of context analysis which are performed automatically by the TWS must be provided by the user under the TBNF system. The three special words (BACK, AHEAD, and NOT), which are used in TBNF to provide contextual information, have no meaning to the BNF half of the TWS. Therefore, when translating into BNF, these special words and the constructs that they herald are meaningless to TWINKLE and are referred to as null symbols. In addition to these null symbols, TWINKLE provides two forms of comments which are meaningless and therefore qualify as null symbols. Any string of symbols enclosed in parentheses, the left-most parenthesis of which is not preceded by either a sharp or a semantic call, constitutes a comment and is deleted by the scanner. Any string of symbols preceded by the special words, COMMENT or C, and not including the symbols [,],;,., or the special words, BEGIN and END, also constitutes a comment.

An empty symbol denotes a string of zero length. It is written as either one of the special words, EMPTY or LAMBDA, or as an adjacent pair of left and right nonterminal parentheses (i.e., < > or " ").

3. CONTROL OF THE TWINKLE TRANSLATION

The TWINKLE translation is merely the first step in a chain of operations undertaken in generating a compiler for a language. Control options specified in the TWINKLE input may be intended for use in a later phase of the TWS. To make the meaning of these options clear a brief description of the entire TWS is now given.

3.1 The Translator Writing System

Figure 2 presents a block diagram of the interrelations between the programs which make up the Translator Writing System when creating a compiler for a language, L. As indicated, the TWS can generate either a recursive descent compiler or a deterministic Floyd production compiler, the decision being made by the user through the PARSER control card. Consideration will be given first to the Floyd production section of the TWS which comprises the TWINKLE translator, the ISL translator (ISLTRAN), BNF2FPL, FPL2PAR, PAR2ALG and finally, the ALGOL compiler.

A unified syntactic and semantic description of L is provided as input to the TWINKLE translator. The translator extracts the syntactic information which it transforms into BNF and places, together with several other tables, it in a disk file labelled L/TABLESF. Similarly, the semantic portion of the input is placed in file, L/ACTIONS, for use by ISLTRAN. The TWINKLE translator then initiates execution of both BNF2FPL and ISLTRAN. The BNF syntax of L is transformed by BNF2FPL into Floyd productions (FPL) which are placed in L/FLOYDP while additional tables are placed in L/TABLESF and the control information in the first record of L/TABLESF is updated. BNF2FPL then initiates execution of FPL2PAR which transforms the FPL syntax from L/FLOYDP into a stream of pseudo-orders which are returned to L/TABLESF. Again, the control information in the first record is updated.

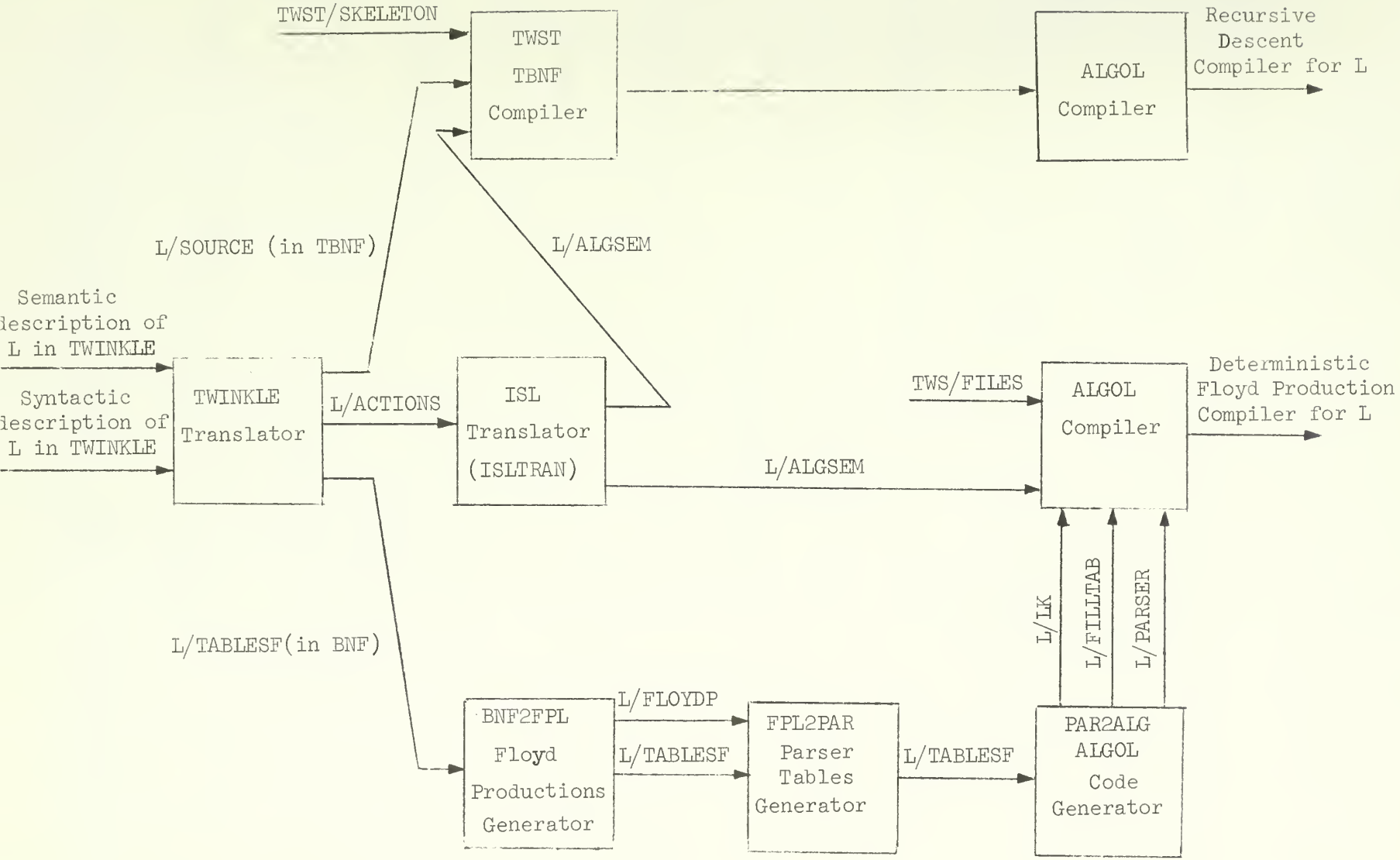


Figure 2. Block diagram of the Translator Writing System.

At this point a small digression must be made to discuss the structure of the parser, itself. The parser consists of three portions: a lookahead section used in the differentiation of Floyd productions by analysis of input symbols as yet unscanned; a section for filling the various parsing tables; and a section which actually performs the parsing. Each of these sections may be either interpretive, operating from the tables in L/TABLESF, or executable (i.e., a self-contained piece of ALGOL code). An executable section must, clearly, be tailor-made for the language, L. Therefore, if any executable sections are requested, FPL2PAR initiates the execution of PAR2ALG, passing to it the requisite information in L/TABLESF. PAR2ALG then creates, as necessary, L/LK, L/FILLTAB, and L/PARSER for executable lookahead, table filling, and parsing, respectively.

Meanwhile, ISLTRAN uses the semantic information in L/ACTIONS to create a package of ALGOL routines to be called by the parser at strategic moments. When these have been assembled in L/ALGSEM, ISLTRAN consults the control information in the first record of L/TABLESF to determine if any executable sections are required. If none is needed, ISLTRAN initiates an ALGOL compilation of L/ALGSEM merged with several standard pieces of code from TWS/FILES. If, on the other hand, one or more executable sections are required for the compiler, ISLTRAN will initiate the compilation only if PAR2ALG has been completed; otherwise, ISLTRAN terminates, the ALGOL compilation being initiated by PAR2ALG upon its completion.

The second half of the TWS, comprising the TWINKLE translator, ISLTRAN, TWST, and the ALGOL compiler, is involved in producing a recursive descent compiler. The TWINKLE translator transforms the syntactic portion of its input into TBNF which it places in L/SOURCE. The semantic portion is collected as before in L/ACTIONS and passed to ISL for processing. Upon completion ISLTRAN initiates execution of TWST which combines the information in L/SOURCE and

L/ACTIONS with the standard basic structure contained in TWST/SKELETON. This is returned to L/SOURCE as an ALGOL version of a compiler for L which is then sent to the ALGOL compiler for compilation--the result being the compiler for L.

3.2 Control Statements

The flow of control described above cannot be wholly automatic. For example, the user must determine, either explicitly or implicitly, whether he wants a Floyd production or a recursive descent compiler. Other user-controlled decisions, while not necessary, are often desirable. The control statements described below provide a large measure of user determined control.

3.2.1 Language Name Designation

The first card in any input deck to the TWS must be the language name designation card. The format of this card is:

LANGUAGE <LANGUAGE NAME>;

where <LANGUAGE NAME> is any string of letters and digits beginning with a letter. These characters, or the first seven if there are more, are used as a prefix for all the interlinking and output files generated by the TWS, including the TWINKLE translator.

3.2.2 Print Options

The print control statement consists of the special word,PRINT, followed by a colon, followed by a list of print options separated by commas, followed by a semicolon. The options available are defined below.

1. TABLES SIFESF: causes the printing of a table displaying the sizes and locations of all of the tables in the file,TABLESF.

2. TERMINALS ALPHABETICALLY, or TRMALF: causes the printing of an alphabetic

list of all of the special words used in the language. If BNF is being generated then the list includes an index of each occurrence of the special words in PROTAB.

3. TERMINALS NUMERICALLY, or TRMNUM: causes the printing of a numerically ordered list of all of the special words used in the language.

4. CHARACTERS, or TRMCHR: causes the printing of an index of the occurrences of the 64 characters in PROTAB.

5. TERMINALS: is equivalent to 2, 3, and 4 taken together.

6. NONTERMINALS ALPHABETICALLY, or NTALF: causes the printing of an alphabetical list of all of the nonterminals used in the language. If BNF is generated, the list includes an index of each occurrence of the nonterminals in PROTAB.

7. NONTERMINALS NUMERICALLY or NTNUM: causes the printing of a numerically ordered list of all of the nonterminals used in the language.

8. NONTERMINALS: is equivalent to 6 and 7 taken together.

9. SYNTAX, or INPUT: causes the printing of the TWINKLE input as it is read.

10. INDEX, or XREF, or CROSS REFERENCE: causes the printing of an index of the occurrences of all nonterminals, terminals, and actions in the syntax by card number.

11. ACTIONS ALPHABETICALLY, or ACTALF: causes the printing of an alphabetical list of all of the semantic actions and tests used in the language. If BNF is being generated, the list includes an index of each occurrence of the actions and tests in PROTAB.

12. ACTIONS NUMERICALLY, or ACTNUM: causes the printing of a numerically ordered list of all of the semantic actions and tests used in the language.

13. ACTIONS: is equivalent to 11 and 12 taken together.

14. PROTAB: is the name of the table into which TWINKLE places the BNF equivalent of TWINKLE syntax in the input. This option causes the printing of this table.

15. FLOYD: BNF2FPL transforms PROTAB into a set of Floyd productions in the disk file, L/FLOYDP. This option causes the printing of these Floyd productions.

16. COMBINED GROUPS: causes the printing of the components of all of the combined groups required by the language. For a discussion on the use of combined groups, see the paper by Beals [3].

17. PARSER: FPL2PAR transforms the Floyd productions in FLOYDP into a stream of pseudo-orders which make up the parser. This option causes the printing of this stream of pseudo-orders.

18. PATTERNS: causes the printing of a table of the <ANY SYMBOL> patterns created in the TWINKLE translator processing of L, as well as any additional patterns created by either BNF2FPL or FPL2PAR.

19. STANDARD: is the union of options 1, 5, 8, 9, 10, 13, 14, and 18.

20. DEBUGGING, or DEBUGN: is the union of options 15, 16 and 19, i.e., of everything but option 17.

21. EVERYTHING: this is the union of all options.

22. NOTHING: this option, when used by itself, inhibits all printing.

If no print control statement appears, the print options are set to the default option, STANDARD.

3.2.3 The Parser Type Option

As mentioned several times above, the TWS is equipped to produce compilers based on either recursive descent or Floyd production language parsers. It is appropriate, therefore, to have a control statement for determining which is to be generated. The relevant control statement is:

<PARSER TYPE> PARSER;

where <PARSER TYPE> is either RECURSIVE DESCENT or FLOYD PRODUCTION.

3.2.4 Zip Control

In Burroughs B-5500 ALGOL it is possible for one ALGOL program to initiate execution of another by executing a zip statement (i.e., by zipping to the other program). The component programs of the TWS use the zip statement to initiate their successors. In normal operation zipping continues through final compilation by the ALGOL compiler. Frequently, a user does not desire execution of the entire TWS but may wish, for example, to check just the syntax, or just the semantics, of the input. This possibility is allowed for in the TWS by the zip control statements which are listed below:

ZIP TO ISL;

DONT ZIP TO ISL;

DONT ZIP;

ZIP THROUGH <PROGRAM NAME>;

where

<PROGRAM NAME> ::= TWST/BNF2FPL/FPL2PAR/PAR2ALG/ISL/ALGOL.

The use of these control statements is self-evident.

3.2.5 Program Parameter Control

Each of the programs in the TWS has certain program parameters which are normally assigned default values that permit compiler generation for many small languages. It is possible, however, that a particular language may require more execution time, a larger stacksize, or a higher B-5500 core memory estimate to run successfully through some phase of the TWS. Correspondingly, it may be desirable when processing some smaller languages to decrease the values of some of these program parameters. This can be done with the three program parameter control statements shown below:

<PROGRAM NAME> PRIORITY = <*N>;

<PROGRAM NAME> CORE = <*N>;

<PROGRAM NAME> STACK = <*N>;

where <PROGRAM NAME> was defined in the last section and <*N> is a positive integer. These set the priority (and, implicitly, the time limit), the core estimate, and the stacksize, respectively, of the program designated. These parameters are then used in zipping to the program. If the <PROGRAM NAME> is COMPILER, the parameters are passed to the ALGOL compiler and become the default parameters for the generated language compiler.

3.2.6 Executable Compiler Options

It was noted in section 3.1 that a Floyd production parser generated by the TWS may be, to a greater or a lesser extent, an executable parser. The default option is a parser which is wholly interpretive, but an executable version of any of the three parser sections may be requested by use of the control statements shown below:

EXECUTABLE LOOKAHEAD;

EXECUTABLE FILL TABLES;

EXECUTABLE FLOYD PRODUCTIONS;

If the lookahead and fill tables portions of the parser are interpretive, the resultant compiler, L/DISK, may only be executed if L/TABLESF is resident on disk. By making these two portions executable, the parser becomes a self-contained unit and compilation in L requires only L/DISK.

3.2.7 Miscellaneous Control Options

CLOSE, CLOSE LP, CLOSE LINEPRINTER, or CLOSE LINE PRINTER: applies to BNF2FPL; it causes a separate file of output to be created each time an error occurs during execution of BNF2FPL. In this way a user can ascertain the cause of some errors before BNF2FPL runs to completion.

LONG LOOKAHEAD: applies to BNF2FPL; it specifies a four symbol lookahead to be used in differentiating before deciding that the group cannot be built. If the Floyd productions of a group being generated cannot be differentiated by a three symbol lookahead and if combination is not possible, the group is not normally built and the BNF2FPL translation fails. In practice it has been found that when a lookahead of three symbols fails, no additional amount of lookahead will help.

COMBINE FIRST: applies to BNF2FPL; it specifies that Floyd production combination be attempted after a one symbol lookahead has failed to differentiate, but before attempting a two or three symbol lookahead; if combination is not possible, two and three symbol lookaheads will be attempted before abandoning the group.

FLOYD PRODUCTIONS PER PROCEDURE: <*N>: applies to PAR2ALG. When creating an executable parser, PAR2ALG generates procedures -- each containing some specified number of the Floyd productions of the language. This number is typically 100, but may be set by the language designer to any desired value.

GROUPS PER PROCEDURE: <*N>: applies to PAR2ALG; determines the number of groups of Floyd productions in each executable parser procedure.

PROGRAM SYMBOL: <X>: is followed by a nonterminal name, say <X>, which is taken to be the unique objective symbol of the language in question; if this option is not used, the first nonterminal to appear in the syntax specification is taken as the unique objective symbol for the language.

SPECIAL SYMBOLS: <SPECIAL WORD LIST>: may be used to force a particular ordering of the special words of the language which are otherwise numbered in the order in which they first appear in the syntax.

3.3 Burroughs B-5500 Control Cards for Executing TWINKLE

The TWINKLE translator is executed like a compiler on the B-5500 system. When the syntax to be translated is on cards, the following deck set up may be used:

```
? USER = Language designer's user code
? COMPILE A/B WITH TWINKLE LIBRARY
? DATA CARD
      input syntax
? END.
```

Since TWINKLE does not create executable code, the file, A/B, is not used, and the name may be specified arbitrarily by the language designer. Because this file is not used, either of the following forms may be used when the input syntax is a file on disk, say PL1/SYNTAX:

```
? USER = Language designer's user code
? COMPILE A/B WITH TWINKLE LIBRARY
? TWINKLE FILE CARD = PL1/SYNTAX SERIAL
? END;
```

or

```
? USER = Language designer's user code
? COMPILE PL1/SYNTAX WITH TWINKLE LIBRARY
? END.
```

In the latter case, TWINKLE discovers that the input is not on cards and that no file has been equated to file, CARD. It then investigates the code file and, if it exists on disk, takes it as the file, CARD. In the former case a file has been equated to file, CARD, so this is taken as the input syntax. In this case, the code file, A/B, is not used and may be named arbitrarily.

4. IMPLEMENTATION OF THE TWINKLE TRANSLATOR

The TWINKLE translator has been implemented with the TWS in a bootstrapping fashion. The preliminary version of the translator was written in BNF and processed on the portion of the TWS then existing, which was essentially equivalent to the BNF2FPL, FPL2PAR and PAR2ALG stages of the current TWS. Each subsequent revision to the TWINKLE translator was implemented with the aid of its predecessor. Thus, although the present syntax is much more sophisticated than the initial syntax,it is also shorter and considerably more readable. The following sections detail some of the salient features of the TWINKLE translator.

4.1 NONTAB, SYMTAB and OPRTAB

As each nonterminal is read from the input syntax, its name is compared against all those presently entered in NONTAB. If a match is found, the corresponding nonterminal number is extracted from the relevant field of the header word for the matching table entry. If no match is discovered, a new entry is made. The entries are linked through the header words in a binary tree which is alphabetically ordered by the nonterminal names. The format of the entries is shown below in figure 3. Associated with each new entry into NONTAB is an entry into NTINDX pointing to the header word of the nonterminal in NONTAB which facilitates printing out the nonterminal names when necessary. Also, if the CROSS REFERENCE print option has been activated by the user, the NTINDX word for a given nonterminal contains a pointer to the base of an inter-linked list of the occurrences of that nonterminal in the input syntax by line number. The actual repository for this list, as well as those for the other nonterminals, terminals, and

action symbols from the input syntax, is an array called OVERALLINDEX, each of whose entries contains a line number, a bit showing whether the specified occurrence was on the right or left hand side of a production, and a pointer to the entry for the next occurrence of the item in whose list the entry resides.

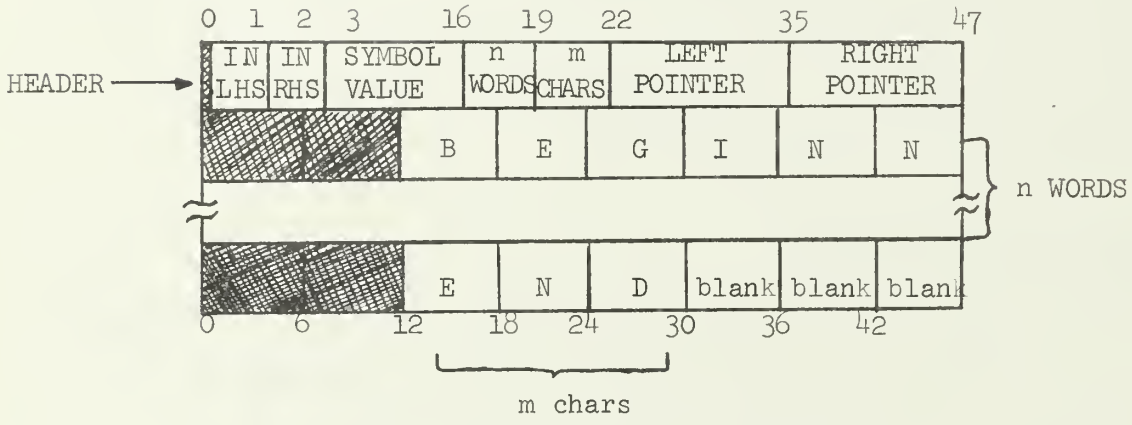


Figure 3. An entry in the NONTAB table

Figure 3 shows the details of a single entry in NONTAB. Consider, first, the header word. Nonterminals (with the exception of the unique objective symbol) used in the syntactic input must appear on the left hand side of at least one production and on the right hand side of at least one (not necessarily different) production. The INLHS bit is set on recognizing the nonterminal as the left hand side of a production and the INRHS bit is set on recognizing it in the right hand side of a production. These bits are checked at the conclusion of syntax input and any discrepancies are reported as errors on the TWINKLE output file, LINE. The SYMBOLVALUE field contains the internal number (code) of the nonterminal symbol represented by this

NONTAB entry. Given a nonterminal name of k characters, n of the WORDS field and m of the CHARS field are given by n = [k/16] and m = k - 6* (n+1). These two fields, taken together, determine the extent of useful information in the remaining words of the NONTAB entry. Finally, the LEFTPOINTER and RIGHT-POINTER fields contain pointers to subsequent entries in the alphabetic binary tree which NONTAB comprises. The remaining words in the NONTAB entry consist of n words containing 6 characters each of the nonterminal name right justified with 2 unused characters at the left, and, in the last word, 2 unused characters, m characters from the nonterminal name, and blanks filled to the right.

Corresponding to NONTAB and NTINDX for nonterminal storage are the pairs of tables (SYMTAB, STINDX), and (OPRTAB, OTINDX) for storing special symbols, and semantic symbols, respectively. As mentioned above, the line by line index information for both these types of symbols is stored in OPERALLINDEX along with that for nonterminals. While STINDX and OTINDX are identical counterparts to NTINDX, entries in SYMTAB and OPRTAB differ slightly from those in NONTAB and, in fact, from one another. In the case of SYMTAB, there is, clearly, no need for the INLHS and INRHS bits since a special symbol, if it appears at all, must appear on the right-hand side of a production. Consequently, in the header words for SYMTAB, these bits are included as a portion of the SYMBOLVALUE field. In OPRTAB header words it is also clear that the INLHS and INRHS bits are unnecessary, but here the first bit is unused and the second bit becomes the USED bit denoting an action or test symbol that has been declared in the syntax. This bit is read by ISL/DISK to determine which actions it must get from the file, <LANGUAGE NAME>/ACTIONS. It is also used by TWINKLE to catch duplicate declarations of the same semantic name.

4.2 PROTAB, PRODS and PDLIST

The primary table into which TWINKLE collects the BNF productions which it produces from its TWINKLE input is PROTAB. Figure 4 shows the fields of a

PROTAB word. The FLAGS field comprises a set of six one bit flags carrying

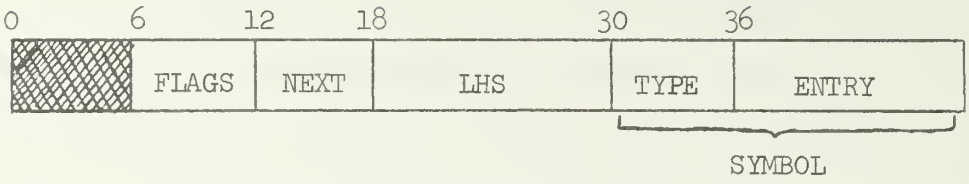


Figure 4. PROTAB word format

various pieces of information. These flags are referred to as: IREC, NOBACK, TRMDER, LASTNT, SFLAG and REC. The IREC flag is set only in the first word of a production and denotes an indirectly left recursive production. That is, a production of the form:

$$\langle A \rangle ::= \langle B \rangle \ \alpha$$

for which <A> is a headsymbol of <B>.

When the NOBACK flag is set, the symbol in the SYMBOL field is not to be back-substituted into this PROTAB location. If the symbol in the SYMBOL field has a terminal derivation, then the TRMDER flag is set. The LASTNT flag is set when the symbol in the SYMBOL field is a nonterminal and, except for possible trailing semantic symbols, is the last symbol in the production. The SFLAG flag heralds a semantic symbol as the next symbol in the production. Finally, the REC flag is set in the first symbol of a production if the symbols in the LHS and SYMBOL fields are the same (i.e., if the production is left recursive). The remaining fields are: the NEXT field which gives the number of words to the beginning of the next production, the LHS field which contains the number of the nonterminal being defined, and the TYPE and ENTRY fields of this particular right-hand side symbol.

TWINKLE performs a number of grammar transformations on a local level. Frequently more than one production is being built simultaneously, as happens when nested definitions are being translated, or when lists are being implemented. These difficulties make it very cumbersome for TWINKLE to put productions directly into PROTAB. To circumvent these problems TWINKLE uses a

256 element entry table (PRODS) as a directory and status table for 255 thirty-two-symbol productions which are stored in the PDLIST array. The words of PRODS form a list linked forward through the NEXTPD fields and backwards through the LASTPD fields. The first entry of PRODS acts as a base for the productions currently in use. The base of available productions is given by the integer variable, FIRSTPDAVAIL. To manipulate these two structures, TWINKLE uses two procedures: GETPROD and GIVEUP. GETPROD is an integer-typed procedure which has no arguments and, when called, returns the address of the next available element of PRODS after incorporating it into the link structure and making the necessary modifications to the various list pointers. GIVEUP has as its sole argument the address of an element of PRODS to be removed from the link structure and returned to available pool. The PDLIST symbols corresponding to PRODS(N) are PDLIST (32 × N) through PDLIST (32 × N + 31).

In addition to serving as the link structure for the productions in PDLIST, each entry of PRODS contains the following information about the production with which it is associated:

(i) COMPLETE: a flag which indicates that the associated production is no longer being extended;

(ii) LEVEL: an eight bit field recording the level of bracket nesting at which the production originated;

(iii) LAMDA: a flag which indicates whether the production has participated in empty context absorption;

(iv) NEXTSYMBOL: a five bit field containing a count of the symbols currently in the production;

(v) DORNO: a three bit field which identifies whether the left-hand side of the production is a simple nonterminal or one of the several TWINKLE generated nonterminal types;

(vi) LHS: a twelve bit field containing the nonterminal number of the

left-hand side of the production.

Each word of PDLIST contains only the left-hand side of the production in the fields, LHSDORNO and LHS, and one symbol from the right-hand side in the fields: DORNO, TYPE, and ENTRY. The remaining information necessary in PROTAB is not filled in until a production actually enters PROTAB.

Productions which are being created in PDLIST may be extended by calling the procedure, ADDON. ADDON has two arguments: the first is the number of the production to be extended; the second is the symbol by which it is to be extended. If the symbol to be added is a TWINKLE-generated nonterminal representing the alternatives of a nested definition set, the procedure for adding it to a production is somewhat complex. Each of the alternatives must be added on individually and, if more than one alternative is present, new productions must be created. Thus, for example, if α represents the string of symbols in the production being extended, and β_1 through β_n represent the strings of symbols in the alternatives of the symbol being added on, ADDON will replace the production:

$$\langle n \rangle ::= \alpha$$

by n productions:

$$\begin{aligned} \langle n \rangle &::= \alpha\beta_1 \\ \langle n \rangle &::= \alpha\beta_2 \\ &\vdots \\ \langle n \rangle &::= \alpha\beta_n \quad . \end{aligned}$$

At the same time ADDON removes the n productions:

$$\begin{aligned} \langle D \rangle &::= \beta_1 \\ \langle D \rangle &::= \beta_2 \\ &\vdots \\ \langle D \rangle &::= \beta_n \end{aligned}$$

where <D> is the TWINKLE-generated nonterminal referred to above. By expanding

nested definitions in this way, TWINKLE ensures that each of the productions which it creates retains all of the context that the language designer specified in the original TWINKLE production.

If the symbol being added on is any other type of symbol, say the symbol, a, ADDON replaces the production:

$$\langle n \rangle ::= \alpha$$

by the production:

$$\langle n \rangle ::= \alpha a$$

where α is as above.

When it has been ascertained that production, P, is completed and is ready to be put into PROTAB, the procedure, PUTINPROTAB, is called with the parameter, P. PUTINPROTAB fills in the NEXT and FLAGS fields of the production and writes it directly into the next available locations in PROTAB. Once this has been accomplished the production is returned to the free pool.

4.3 PRODSTACK and LPSTACK

The TWINKLE language offers the user only two recursive constructs. These are nested definitions and list structures, which are implemented with PRODSTACK and LPSTACK, respectively. These are dimensioned to allow nesting of either definitions or lists to a depth of thirty, but this may, of course, be easily altered in the unlikely event that it needs to be. The formats of words in these two stacks are shown below (in figures 5 and 6, respectively).

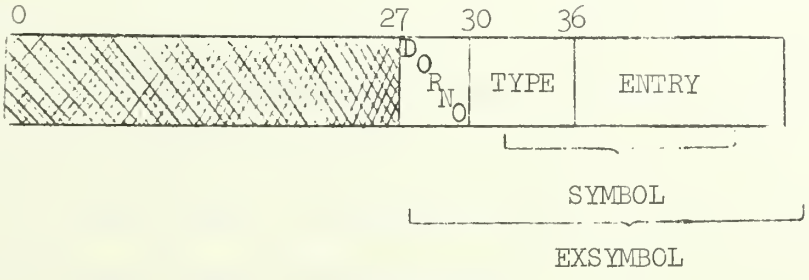


Figure 5. The format of a PRODSTACK entry.

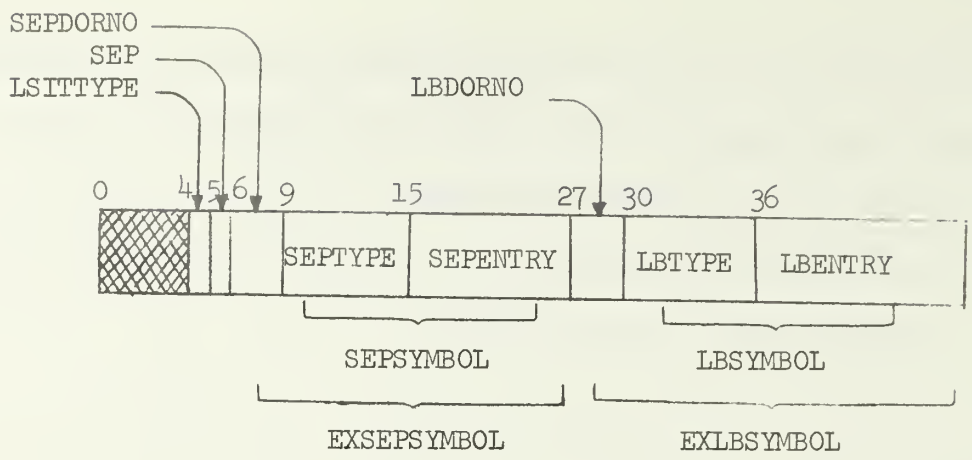


Figure 6. The format of a LPSTACK entry.

PRODSTACK is simply a stack of single symbols--the top symbol being the left-hand side of the set of productions currently being built. Whenever the left-hand side of a TWINKLE production is encountered, the left-hand side nonterminal symbol is pushed into PRODSTACK. Similarly, when the left square bracket of a square brackets construct is encountered, a TWINKLE-generated temporary dummy (the DORNO field is set to one) is created and pushed into PRODSTACK. Right square brackets and semicolons, which end square bracket constructs and productions, respectively, cause the top of PRODSTACK to be popped. PRODSTACK is used in assigning names to TWINKLE-generated permanent dummy nonterminals in the following manner. When such a dummy is required (e.g., in the generation of lists, see below), PRODSTACK is searched downward from the top for a natural nonterminal (which is identified by a zero in the DORNO field). There must always be such an entry because PRODSTACK always extends to the beginning of some TWINKLE production which must start with a natural nonterminal. The alphanumeric characters which make up the name of the nonterminal are obtained from NONTAB. The desired name is then created by adding to these characters a blank, the characters "DUMMY" another blank, and finally a number unique to this nonterminal. Since blanks cannot appear in natural nonterminal names, these serve to ensure that no duplication of

nonterminal names can arise by this procedure. As an example of this, the list structure in the TWINKLE production:

A <PROGRAM> CONSISTS OF A LIST OF <STATEMENT>S;

would be implemented with a permanent dummy nonterminal named PROGRAM DUMMY 1.

Each entry of LPSTACK eventually contains all the information necessary to construct the BNF equivalent productions for the list structure which generated it. Whenever the list type of a list structure is encountered, a new word is pushed into LPSTACK with an appropriate setting of the LISTTYPE bit (i.e., 1 for a nonempty list and 0 for a possibly empty list). When a list base is recognized, the sub-fields of the EXLBSYMBOL field are set in the top word of LPSTACK to identify the base. Similarly, recognition of a list separator causes the subfields of the EXSEPSYMBOL to be filled in the top word of LPSTACK; the SEP bit is set to zero for a definite separator and to one for a possibly empty separator. If the list does not have a separator, the EXSEPSYMBOL field is set to indicate an empty separator and the SEP bit is set to zero.

The LPSTACK entry does not reflect the type of recursive desired for the list, i.e., left recursive or right recursive. However, this is determined syntactically and is transmitted to the semantics via the choice of the action called. Given that EXSEPSYMBOL contains the symbol, S, and that EXLBSYMBOL contains the symbol, B, figure 7 shows the productions which are generated to implement the list for various choices of SEP and LISTTYPE where <RD> is the TWINKLE-generated permanent dummy which implements the list structure and <XD> is a TWINKLE-generated temporary dummy. Recall that a SEP bit of one designates a definite separator and a zero bit designates a possibly empty separator; whereas, a LISTTYPE bit of one indicates a nonempty list and a zero bit indicates a possibly empty list.

LISTYPE	0	0	1	1	
SEP	0	1	0	1	
	yes	yes	no	no	<XD> ::= <RD>
	yes	yes	no	no	<XD> ::= < >
	yes	yes	yes	yes	<RD> ::= <RD> S B
	yes	no	yes	no	<RD> ::= <RD> B
	yes	yes	yes	yes	<RD> ::= B

(a) Left

LISTYPE	0	0	1	1	
SEP	0	1	0	1	
	yes	yes	no	no	<XD> ::= <RD>
	yes	yes	no	no	<XD> ::= < >
	yes	yes	yes	yes	<RD> ::= B S <RD>
	yes	no	yes	no	<RD> ::= B <RD>
	yes	yes	yes	yes	<RD> ::= B

(b) Right

Figure 7. Left recursive and right recursive lists.

Note that possibly empty lists are characterized by the <XD> nonterminal; whereas, nonempty lists are characterized directly by the list implementing nonterminal <RD>.

4.4 Any Patterns

An any pattern comprises 384 bits (eight words of 48-bits each) for which there exists a one-to-one correspondence between the lower numbered bits and the terminal symbols of the language. Bits zero through 63 correspond to the 64 characters of the Burroughs B5500 system; bits 66 through 85 correspond to the possible special terminal classes; bits 86 and above correspond to the special words of the language. Bits 64 and 65 are never set in an any pattern because they correspond to a terminating character and an illegal character, respectively. Any patterns are stored end-to-end in a 512-word array, ANYPAT. Each bit that is set in an any pattern indicates that the corresponding terminal is represented by the any pattern.

During the preliminary translation, any patterns are actually created in the negative--that is, the bits are set if the corresponding terminal is not represented by the particular any pattern. This condition is rectified when the syntax input has been completed. Each pattern is transmitted to the procedure, CLEANUP, which converts it to the required form which may involve recursive calls of CLEANUP if the any base for the pattern was a nonterminal.

4.5 Grammar Transformations

The PROTAB generated by a TWINKLE translation is not, in general, in a form acceptable to BNF2FPL. A number of transformations must therefore be performed on a BNF grammar to increase the probability of its acceptance to the overall TWS. In addition, a few transformations are applied to increase the efficiency of the resultant compiler. These are described below in the order in which they are performed.

4.5.1 Back Context Absorption

In translating from TWINKLE into BNF, it is important that TWINKLE retain any context in the BNF that was inherent in the original TWINKLE

production. When a left recursive list is created, it is necessary for the non-recursive productions of the list-implementing nonterminal to absorb any of the context prior to it that may have arisen from constructs within the TWINKLE production that initially gave rise to the list. As may perhaps be anticipated, problems begin to crop up when more than one list is included in a single TWINKLE production--as in the case of nested lists.

Each time a definition is complete (i.e., whenever a slash, right square bracket, semicolon, or special word, OR, is encountered), the following context-absorbing algorithm is invoked. Let $\{P_n | n = 1,\ldots, N\}$ be the set of productions in PDLIST and $\{D_n | n = 1,\ldots, N\}$ be the corresponding entries in PRODS. Further, let $P_n(j)$ denote the j-th symbol on the right-hand side of the n-th production and let R_n denote the corresponding left-hand side. The right-hand sides of P_n are then scanned for the occurence of a left-recursive TWINKLE-generated dummy nonterminal, say $P_n(j)$ (such a nonterminal is characterized by a 3 in the DORNO field discussed in section 4.2). If $j > 1$, or if the LAMDA bit of PRODS is set, a new production, P_k $(k = N + 1)$, is created as follows:

$$P_k(i - j + 1) = P_n(i) \quad i = j, j + 1,\ldots, \ell_n,$$

where ℓ_n is the number of symbols in P_n. The LAMDA bit of D_k is set to zero.

The following steps are then taken for each production P_k, for which $k \leq N$ and $R_k = P_n(j)$ while $R_k \neq P_k(1)$:

(i) the LEVEL field of D_k is set to OLDMARKER;

(ii) k is incremented and a new production, P_k, is constructed according to:

$$P_k(i) = P_n(i) \qquad i = 1, 2,\ldots j-1$$

$$P_k(j-1+i) = P_k(i) \quad i = 1, 2, \ldots, \ell_k;$$

(iii) the LAMDA bit of D_k is set to one;

(iv) $R_k = R_k$.

Finally, P_n is removed and scanning continues, skipping over productions, P_m, in which the LEVEL field of D_m is set to OLDMARKER, until all productions, including those created above, are exhausted.

There is one circumstance under which P_k, above, does not participate in context absorption. This is the case in which both R_n and $P_k(1)$ are left-recursive dummies and where among the $\{P_n\}$, R_n is a head symbol of R_k. In this case, context absorption will lead to an infinite loop, absorbing alternately a R_k and a R_n.

The following example provides an illustration of context absorption applied to the TWINKLE production:

<A> ::= a LIST [LIST b SEP c] d LIST e

where a, b, c, and d represent terminal symbols. The productions, before context absorption, are:

P_1: <A> ::= λa<RD1> d <RD3>

P_2: <RD1> ::= λ<RD2>

P_3: <RD1> ::= <RD1> <RD2>

P_4: <RD2> ::= λb

P_5: <RD2> ::= <RD2> c b

P_6: <RD3> ::= λe

P_7: <RD3> ::= <RD3> e

where λ denotes productions, P_i, for which the LAMDA bit of D_i is set to 1. Then:

P_2: <RD1> ::= λ <RD2> *

P_4: <RD2> ::= λ b *

P_5: <RD2> ::= <RD2> c b

P_6: <RD3> ::= λ e

P_7: <RD3> ::= <RD3> e

P_8: <A> ::= <RD1> c <RD3>

P_9: <RD1> ::= λ a<RD2>

P_{10}: <RD1> ::= <RD2>

P_{11}: <RD2> ::= <RD1> b

are the productions remaining after $P_1(1)$ and $P_3(2)$ have participated in context absorption. Note that $P_2(1)$ does not participate because the asterisk indicates that the LEVEL field of D_2 is equal to OLDMARKER. Since $P_5(1)$ and $P_7(1)$ cannot participate because they lack a λ:

P_2: <RD1> ::= λ <RD2> *

P_4: <RD2> ::= λ b *

P_5: <RD2> ::= <RD2> c b

P_6: <RD3> ::= λe *

P_7: <RD3> ::= <RD3> e

P_{10}: <RD1> ::= <RD2>

P_{11}: <RD2> ::= <RD1> b

P_{12}: <A> ::= <RD3>

P_{13}: <RD3> ::= <RD1> c e

P_{14}: <RD2> ::= λ a b

show the productions remaining after context has been absorbed from $P_8(3)$ and $P_9(2)$. Note that the duplication of P_{10} from P_9 is inhibited. Finally:

P_5: <RD2> ::= <RD2> c b

P_7: <RD3> ::= <RD3> e

P_{10}: <RD1> ::= <RD2>

P_{11}: <RD2> ::= <RD1> b

P_{12}: <A> ::= <RD3>

P_{13}: <RD3> ::= <RD1> c e

P_{14}: <RD2> ::= λ a b

show the results of eliminating all productions marked with an asterisk. Note also that RD1 is no longer explicitly left-recursive but retains implicit left-recursiveness through P_{10} and P_{11}. It is easy to see that the terminal strings defined by the fast set of productions above are exactly those defined by P_1 through P_7 originally.

Context absorption is a local grammar transformation and is, therefore, carried out in PDLIST before the productions are entered into PROTAB. All of the grammar transformations are global and are performed in PROTAB, itself. To discuss these with some facility, the notation of this section will be altered and augmented as follows. The i-th production in PROTAB will be denoted by P_i, its right-hand side symbols by $P_i(j)$, where j ranges from 1 through ℓ_i, the number of symbols on the right-hand side. The left-hand side will be denoted by R_i. Finally, for each nonterminal n, N(n) and T(n) will be the sets of nonterminal and terminal head symbols of n, respectively.

4.5.2 Empty Removal

The control of ADDON and PUTINPROTAB is such that empty symbols may appear in PROTAB only as productions in themselves. That is, if a production contains an empty symbol, that must be its only symbol. Even in this relatively mild form, however, empty symbols are unacceptable to BNF2FPL and it falls to TWINKLE to remove them by back-substituting and collapsing PROTAB around them. Since PROTAB always contains the initial production:

<WHOLE PROGRAM> ::= ⊥ <PROGRAM SYMBOL> ⊥

where <PROGRAM SYMBOL> is the unique objective symbol of the language, and "⊥" is a special termination terminal symbol that appears nowhere else in PROTAB. Furthermore, there is no nonterminal for which the only production is LAMBDA since there is a check that each nonterminal has a terminal string derivation.

Each production, P_i, for which $\ell_i = 1$ is tested to determine whether $P_i(1)$ is the empty symbol, LAMBDA. When such a production is found, PROTAB is scanned for occurrences of R_i. Such an occurrence, say $P_n(j) = R_i$, generates a new production, P_k, at the end of PROTAB according to the rules:

(i) if $\ell_n = 1$ then

a) $\ell_k = 1$

b) $R_k = R_n$

c) $P_k(1)$ is an empty symbol;

(ii) if $\ell_n > 1$ then

a) $\ell_k = \ell_n - 1$

b) $R_k = R_n$

c) $P_k(m) = P_n(m)$ $m=1,\ldots,j-1$; $P_k(m)=P_n(m+1)$, $m=j,\ldots,\ell_k$.

In the first case, P_k will be picked up in its turn as an empty production. In the second case, P_k will eventually be scanned for occurrences of R_i and possibly generate further new productions. During the course of this procedure new productions are compared against existing productions in PROTAB for identity. If a match is found the new production is not entered into PROTAB.

4.5.3 Back Substitution of Singly Defined Nonterminals

Unlike LAMBDA removal, without which PROTAB is unacceptable to BNF2FPL, back-substitution of singly-defined nonterminals merely serves to

increase the overall efficiency of the resultant compiler by decreasing the number of reductions required to recognize certain nonterminals.

The algorithm for back-substitution is very straightforward and proceeds as follows. The set, A, of singly defined nonterminals is determined by one pass through PROTAB. Then, for each nonterminal, n, in A, PROTAB is scanned for productions, P_i, in which $P_i(j) = n$ for some j such that $1 \leq j \leq \ell_i$. A new production, P_k, is then created such that:

(i) $R_k = R_i$;

(ii) $\ell_k = \ell_i + \ell_{n'} - 1$;

(iii) $P_i(m)$ $m = 1,\ldots, j - 1$;

$P_{n'}(m - j + 1)$ $m = j,\ldots,j + \ell_{n'} - 1$;

$P_i(m - \ell_{n'} + 1)$ $m = j + \ell_{n'},\ldots,\ell_k$,

where $P_{n'}$ is the single production for which $R_{n'} = n$. The production, P_i, is then deleted and the scan for occurrences of n continues, eventually reaching P_k and checking for possible further occurrences of n.

4.5.4 Dummy Insertion

It has been mentioned, briefly, that two otherwise indistinguishable, Floyd productions may be differentiated by a look ahead of at most three symbols. If this much look ahead is insufficient, BNF2FPL attempts to combine the productions deciding (in essence) that the differentiation may be postponed. Of course differentiation must eventually be accomplished by look ahead, if at all. There are two situations in which this combination cannot be performed. First, a terminal symbol may not be combined with a nonterminal. Second, if A and B are two nonterminals for which a combination is proposed, that combination cannot be carried out if A is a head symbol of B (or, of course, if B is a head symbol of A). The reason for this is that the parser, in the course of

looking for an A or a B, will then be satisfied by finding an A, even though that A may, in fact, be the beginning of a B which a correct parse would discover.

The approach to this problem in the original TWS was to look for BNF productions of the form:

$$\langle N_1 \rangle ::= \alpha\ A\ \beta \qquad (1)$$
$$\langle N_2 \rangle ::= \alpha\ \langle B \rangle\ \gamma$$

where α is a nonempty string of terminals and/or nonterminals, and A is a headsymbol of the nonterminal, B (A may be either a terminal or a nonterminal). These productions were modified to:

$$\langle N_1 \rangle ::= \alpha\ \langle Q \rangle \beta$$
$$\langle N_2 \rangle ::= \alpha\ \langle B \rangle\ \gamma \qquad (2)$$
$$\langle Q \rangle ::= A \quad .$$

Now Q is not a headsymbol of B and, if a Q is found, there can be no question of its actually heralding a B.

This can create problems, however, when $\beta = \beta_1\beta_2$ and there exists a production:

$$\langle N_3 \rangle ::= A\beta_1\eta \quad . \qquad (3)$$

If β_1 is more than three symbols long, the parser is unable to determine, when it finds an A, whether that A should become a Q or a N_3. It is not possible to combine since the Q production calls for an immediate reduction.

To obviate this difficulty, the present system performs dummy insertion (Q is referred to as a dummy) by replacing the productions (1) by:

$$\langle N_1 \rangle ::= \alpha\ \langle Q \rangle$$
$$\langle N_2 \rangle ::= \alpha\ \langle B \rangle$$
$$\langle Q \rangle ::= A\beta_1\beta_2$$

instead of the productions (2). The production (3) now causes no problem (if η and β_2 are differentiable) because the burden of the differentiation can be put off by combination of the strings β_2 and η . Experience has shown that this modification has profound effect on the ease of writing acceptable grammars.

The algorithm by which TWINKLE performs dummy insertion is based on two procedures, MAKEDUMMYS and FIX, which call one another recursively. MAKEDUMMYS has three arguments (BASIC, FIRST, and LAST) which are addresses of productions in PROTAB. Each production from FIRST to LAST is compared against the production at BASIC to see if a dummy is needed. If it is determined that a dummy is required at PROTAB [I], FIX is called with I as a parameter and proceeds to make the necessary alterations to PROTAB. The production thus created may, in turn, require dummies when compared to productions prior to BASIC. To make these comparisons, FIX calls MAKEDUMMYS before returning.

4.6 TWINKLE Output Files

4.6.1 TABLESF

The TABLESF file is the basic thread that ties the TWS together. The first record of TABLESF contains control information concerning which options are in effect, a directory of the various tables appearing in later records, and a brief of the progress through the system. This latter contains the last program to run successfully and the dates and times at which each program was most recently executed. Below is a list of entries in the first record of TABLESF.

LANGNAME - the name of the language truncated to seven characters;

TABLEBEGIN - a pointer to the first word of the table directory;

PRINTOPTIONS - a word of control bits for printing, and a few other options;

LASTPROGRAM - the number of the last program to run successfully

(0 = TWINKLE; 1 = BNF2FPL; 2 = FPL2PAR; 3 = PAR2ALG);

ZIPTHROUGHBITS - the number of the program through which the user desires the system to zip automatically;

FIRSTGROUP - used by BNF2FPL;

TWINKLEDATE
TWINKLETIME
BNFDATE
BNFTIME
FPLDATE
FPLTIME
PARDATE
PARTIME

the date and time at which the indicated program was last run successfully;

FPLPRIORITY
FPLCORE
PARPRIORITY
PARCORE
ISLPRIORITY
ISLCORE
ISLSTACK
ALGOLPRIORITY
ALGOLCORE
ALGOLSTACK
COMPPRIORITY
COMPCORE
COMPSTACK
TABPRIORITY
TABCORE
TABSTACK

these are the various user controlled execution parameters of the programs indicated; the prefix, COMP, refers to the resultant compiler and the prefix, TAB, refers to the TWS printing program, PRINTAB/TWS;

NOSYM - the number of special terminal classes required by the language;

NONDUM - the number of nonterminals prior to dummy insertion;

FDDERPROC - the number of Floyd productions per procedure as set by the control card described in section 3.2.6;

STKTOP - the length of the longest production in PROTAB;

NTS - the number of special words used in the language;

TINDEXSTART - the record in which the special word index table, STINDX, begins;

NNS - the number of nonterminals used in the language;

NTINDEXSTART - the record in which the nonterminal index table, NTINDX, begins

NOS - the number of actions and tests used in the language;

OPRINDEXSTART - the record in which the action index table, OTINDX, begins;

SPSTABPT - the number of entries in SYMTAB;

SPSTABSTART - the record in which SYMTAB begins;

NONTABPT - the number of entries in NONTAB;

NONTABSTART - the record in which NONTAB begins;

OPRTABPT - the number of entries in OPRTAB;

OPRTABSTART - the record in which OPRTAB begins;

PROTABPT - the number of entries in PROTAB;

PROTABSTART - the record in which PROTAB begins;

PATTERNLENGTH - the length of each any pattern;

PATTERNPT - the number of entries in ANYPAT;

PATTERNSTART - the record in which ANYPAT begins;

NOMORETABPT - a dummy pointer;

NOMORETABSTART - the first available record at the end of ANYPAT.

By using TABLEBEGIN and record pointers for referencing resident tables, the format of TABLESF is made very flexible. In fact, several additional tables are placed in TABLESF as a language runs its course through the TWS. The LASTPROGRAM entry ensures that no program in the TWS will operate on a TABLESF file that has not been properly prepared.

4.6.2 ACTIONS

Although semantic information is also included in the TWINKLE input, this information is placed, virtually without any processing, into the file, ACTIONS, which is passed on to the ISL translator for the majority of its processing. ACTIONS is a file of card images of which the first contains the number of different actions and tests used in the syntax, and the card number on which the semantic tail, if any, begins. If there is no semantic tail, this latter number is zero.

Each action or test name is written onto the next available card in ACTIONS when it is read for the first time. When a semantic declaration call is encountered, the code which defines it is placed on consecutive cards in ACTIONS. To set these apart from the action and test names, two special cards are emitted before the code and two after it. The first card emitted has dollar signs in the first four columns and the number of the action with which the test is associated following these. The second card has only the special word, BEGIN. The first card following the code has only the special word, END, and, finally, there is a card with dollar signs in the first four columns once again.

Implicit actions are treated similarly with the exception that a name must be generated for these and placed in ACTIONS along with the code. The

names generated for these actions are IMPLICIT1, IMPLICIT2, etc.

4.7 ZIP Files

TWINKLE must have the facilities for initiating the execution of several other programs, depending on the various control parameters obtained from the input. Since each program is initiated in essentially the same way, the following description of the initiation of BNF2FPL should suffice as an illustration.

When it has been determined that a zip to BNF2FPL/TWS is required, the translator executes a SEARCH on file BNF2FPL, the multifile- and file-identifiers of which are "BNF2FPL" and "TWS", respectively. This determines whether or not BNF2FPL/TWS is presently in the disk directory. If it is not, the following message is issued to the system SPO:

```
#TWINKLE SUSPENDED:
 BNF2FPL/TWS NOT IN DIRECTORY
 OK ZIP:  mAXOK
 NO ZIP:  mAXNO
```

where, m is the mix number of the TWINKLE translator. At this point, the operator must either load BNF2FPL/TWS and then restore TWINKLE to execution by entering mAXOK at the SPO, or terminate the TWINKLE execution by entering mAXNO at the SPO.

Assuming that BNF2FPL was present or has been loaded, TWINKLE writes the following control cards in BNFTEMPFILE:

```
? EXECUTE BNF2FPL/TWS; FILE NAME = Xaaaaaa
? STACK = nnnn
? DATA Xaaaaaa
aaaaaaa
? END
```

5. SUMMARY

The TWINKLE metalanguage unifies the TWS by accepting, virtually without alteration, syntax specifications written in either TBNF or BNF and generating either TBNF or BNF specifications which then may be used to create either a recursive descent or a deterministic Floyd production parser, respectively. The changes necessary to be made to the input involve such things as additional sharps on special words and characters, which frequently may be accomplished after a single preliminary run through the translator.

TWINKLE goes well beyond these sub-languages, however, by offering a rich syntax for creating readable English descriptions of languages which are directly acceptable for computer manipulation. Thus, TWINKLE achieves the "lean mix of compact syntax metalanguage with natural language" which Perstein [2] has advocated. The sacrifice for this generality has been that the TWINKLE language, itself, is much more difficult to describe than either BNF or TBNF. Possible ambiguities arise in the use of English; the interpretation taken must be stated explicitly. It is felt, however, that the TWINKLE interpretation is most frequently that which would naturally occur to native speakers of the English language and hence, although precise definition requires much more effort, one's familiarity with natural English makes the TWINKLE syntax of a language immediately understandable.

Floyd [12] offers the following example of BNF in explaining its metasymbolism:

```
<for statement> ::= <for clause> <statement>
                  | <label> : <for statement>
```

which, he says, "can be read 'A for-statement is defined to be a for-clause followed by a statement or a label followed by a colon followed by a for-statement.'" With the exception that TWINKLE retains the angle brackets <and>,

this latter form is also perfectly valid TWINKLE. But later in his paper Floyd remarks:

> "It is evident that a complete definition of a programming language may be expressed far more concisely by a PSG [Phrase Structure Grammar] than by the corresponding English sentences and that it is humanly impossible to write those sentences, with their hundreds of occurrences of 'is defined to be,' 'followed by,' and 'or'."

While the conciseness of BNF is only marginally debatable, it is clear, from Appendix B, that the English syntax of TWINKLE was neither impossible to write nor glutted with hundreds of these phrases.

Finally, the use of TWINKLE constructs, such as lists and enclosures, allows TWINKLE to create a BNF syntax which has a significantly higher probability of acceptance by the TWS than the BNF which the language designer would have created on his own. This is accomplished through grammar transformations, such as back context absorption, of which the language designer may himself be capable, yet which are so tedious and error prone that he is loathe to undertake them.

APPENDIX A

RESERVED WORDS FOR TWINKLE

ACTALF
ACTIONS
ACTNUM
AHEAD
ALGOL
ALPHABETICALLY
ALPHA
AMPERSAND
AMPERSANDS
AND
ANGLE
AN
ANY
ARROW
ARROWS
A
ASTERISK
ASTERISKS
BACK
BEGINNING
BEGIN
BE
BIT
BLANK
BNF2FPL
BRACKET
BRACKETS
BUT
BY
CHARACTER
CHARACTERS
CLOSE
CODE
COLON
COLONS
COMBINED
COMMA
COMMAS
COMMENT
COMPILER
CONSISTS
CORE
CROSS
C
DEBUGGING
DEBUGN
DEFINED
DESCENT
DIGIT
DIGITS
DOLLAR
EITHER
EL
EMPTY
ENCLOSED
END
EQUAL
EVERYTHING
EXECUTABLE
EXPANDING
FILL
FIRST
FLOYD
FOLLOWED
FPL2PAR
GREATER
GROUPS
IDENTIFIER
IDENTIFIERS
INDEX
INPUT
IN
I
ISL
IS
KLEENE
LAMBDA
LANGUAGE
LEFT
LESS
LETTER
LETTERS
LINEPRINTER
LINE
LIST
LONG
LOOKAHEAD
LP
L
MARK
MARKS
NOBACK
NONCHARACTER
NONCHARACTERS
NONEMPTY
NON
NONTERMINALS
NOTHING
NOT
N
NTALF
NTNUM
NUMBER
NUMBERS
NUMERICALLY
OFF
OF
ONE
ON
OPEN
OPERATOR
OPERATORS
OR
PAR2ALG
PARENTHESES
PARENTHESIS
PARSER
PATTERNS
PERCENT
PERIOD
PERIODS
PER
POSSIBLY
PRINTER
PRINT
PRIORITY
PROCEDURE
PRODUCTION
PRODUCTIONS
PROGRAM
PROTAB
Q
QUESTION
QUOTE
QUOTES
RECURSIVE
REDUCED
REFERENCE
RELATIONAL
RESET
RIGHT
R
SEMICOLON
SEMICOLONS
SEPARATED
SEPARATOR
SEP
SEQUENCE
SET
SHARP
SHARPS
SIGN
SIGNS
SIZES
SLASHES
SLASH
SPECIAL
SQUARE
S
STACK
STANDARD
STAR
STARS
STARTING
STRING
STRINGS
SYMBOL
SYMBOLS
SYNTAX
TABLES
TABLESF
TERMINAL
TERMINALS
THAN
TO
TRMALF
TRMCHR
TRMNUM
T
TWST
VIRGULE
VIRGULES
WHICH
WITH
XREF

APPENDIX B

```
A <TWINKLE PROGRAM> CONSISTS OF A <LANGUAGE SYNTACTIC DESCRIPTION>  *   100
  FOLLOWED BY POSSIBLY ONE [SEMICOLON OR #END] FOLLOWED BY POSSIBLY *   200
  ONE <SEMANTIC TAIL>;                                              *   300
A <LANGUAGE SYNTACTIC DESCRIPTION> CONSISTS OF A LIST OF <CONTROL    *   400
  STATEMENT>S SEPARATED BY SEMICOLONS STARTING WITH A <LANGUAGE     *   500
  NAME DESIGNATION> FOLLOWED BY A SEMICOLON FOLLOWED BY A LIST OF   *   600
  <PRODUCTION>S SEPARATED BY SEMICOLONS;                            *   700
A <PRODUCTION> IS DEFINED TO BE AN [<ARTICLE> WHICH IS EITHER [AN    *   800
  #A OR AN #AN OR EMPTY]] FOLLOWED BY A <NONTERMINAL> FOLLOWED BY   *   900
  AN <ARROW> FOLLOWED BY A NONEMPTY LIST OF <DEFINITION>S SEPARATED *  1000
  BY EITHER [A SLASH OR AN #OR];                                    *  1100
AN <ARROW> IS EITHER [COLON COLON EQUAL SIGN  OR #IS OR #IS #DEFINED*  1200
  #TO #BE OR #CONSISTS #OF];                                        *  1300
A <DEFINITION> CONSISTS OF A LIST OF <SYMBOL>S POSSIBLY SEPARATED   *  1400
  BY [#FOLLOWED #BY <ARTICLE>] BEGINNING WITH AN [<ARTICLE> FOLLOWED*  1500
  BY A <SYMBOL>];                                                   *  1600
A <SYMBOL> IS EITHER [A <SYNTACTIC SYMBOL> OR A <SEMANTIC SYMBOL>   *  1700
  OR A <NULL SYMBOL> OR AN <EMPTY SYMBOL>];                         *  1800
A <SYNTACTIC SYMBOL> CONSISTS OF EITHER [A <SIMPLE SYNTACTIC SYMBOL>*  1900
  FOLLOWED BY AN [AMPERSAND OR A QUESTION MARK]    OR POSSIBLY ONE  *  2000
  [#POSSIBLY #ONE] FOLLOWED BY A <GENERAL SYNTACTIC SYMBOL>];       *  2100
A <GENERAL SYNTACTIC SYMBOL> CONSISTS OF EITHER [A <SEEDED LIST> OR *  2200
  A <SIMPLE SYNTACTIC SYMBOL>] FOLLOWED BY A POSSIBLY EMPTY LIST OF *  2300
  <ENCLOSURE OPERATOR>S;                                            *  2400
A <SIMPLE SYNTACTIC SYMBOL> IS A <TERMINAL> OR AN <ANY SYMBOL> OR   *  2500
  A <NONTERMINAL> OR POSSIBLY ONE #EITHER FOLLOWED BY EITHER        *  2600
  [A <SQUARE BRACKET CONSTRUCT> OR A <NESTED PRODUCTION>] FOLLOWED  *  2700
  BY POSSIBLY ONE <KLEENE STAR>;                                    *  2800
A <SQUARE BRACKET CONSTRUCT> IS A LIST OF <DEFINITION>S SEPARATED   *  2900
  BY EITHER [A SLASH OR #OR] ENCLOSED IN SQUARE BRACKETS;           *  3000
A <NESTED PRODUCTION> IS [AN <ARTICLE> FOLLOWED BY A <NONTERMINAL>  *  3100
```

```
   FOLLOWED BY POSSIBLY ONE #WHICH FOLLOWED BY AN <ARROW> FOLLOWED   *  3200
   BY A LIST OF <DEFINITION>S SEPARATED BY EITHER [SLASH OR #OR]]    *  3300
   ENCLOSED IN SQUARE BRACKETS)                                      *  3400
A <KLEENE STAR> CONSISTS OF POSSIBLY ONE [#CLOSE OR #OPEN] FOLLOWED  *  3500
   BY EITHER [A STAR OR #+])                                         *  3600
AN <ENCLOSURE OPERATOR> CONSISTS OF #ENCLOSED #IN FOLLOWED BY EITHER*  3700
   [#PARENTHESES OR #SQUARE #BRACKETS OR #ANGLE #BRACKETS OR A       *  3800
   <SIMPLE SYNTACTIC SYMBOL> FOLLOWED BY POSSIBLY ONE #S FOLLOWED BY *  3900
   POSSIBLY ONE [#AND <SIMPLE SYNTACTIC SYMBOL> FOLLOWED BY POSSIBLY *  4000
   ONE #S]])                                                         *  4100
A <SEEDED LIST> IS EITHER [A <LEFT RECURSIVE LIST> OR A <RIGHT       *  4200
   RECURSIVE LIST>] FOLLOWED BY POSSIBLY ONE [<LIST SEED> WHICH IS   *  4300
   DEFINED TO BE EITHER [#STARTING OR #BEGINNING] #WITH <ARTICLE>    *  4400
   FOLLOWED BY EITHER [A <SIMPLE SYNTACTIC SYMBOL> OR A <SEEDED      *  4500
   LIST>]])                                                          *  4600
AN <ANY SYMBOL> CONSISTS OF AN <ANY BASE> FOLLOWED BY                *  4700
   POSSIBLY ONE <EXCEPTION>)                                         *  4800
AN <ANY BASE> IS DEFINED TO BE #ANY FOLLOWED BY [A <NONTERMINAL>     *  4900
         OR #DIGIT                    OR #DIGITS                    *  5000
         OR #LETTER                   OR #LETTERS                   *  5100
         OR #SPECIAL #CHARACTER       OR #SPECIAL #CHARACTERS       *  5200
         OR #CHARACTER                OR #CHARACTERS                *  5300
         OR #RELATIONAL #OPERATOR     OR #RELATIONAL #OPERATORS     *  5400
         OR #NON #CHARACTER           OR #NON #CHARACTERS           *  5500
         OR #NONCHARACTER             OR #NONCHARACTERS             *  5600
         OR #TERMINAL                 OR #TERMINALS       ])        *  5700
AN <EXCEPTION> CONSISTS OF #BUT FOLLOWED BY EITHER [A LIST OF        *  5800
   <TERMINAL>S SEPARATED BY #BUT  OR A LIST OF <TERMINAL>S POSSIBLY  *  5900
   SEPARATED BY EITHER [SLASHES OR #OR]  ENCLOSED  IN  SQUARE        *  6000
   BRACKETS])                                                        *  6100
A <TERMINAL> IS DEFINED TO BE EITHER [A <SYMBOLIC TERMINAL> OR AN    *  6200
```

```
  <ENGLISH TERMINAL>]}                                                  *  6300
A <SYMBOLIC TERMINAL> IS EITHER [#BLANK OR A <TERMINAL CLASS NAME>       *  6400
  ENCLOSED IN <LEFT BRACKET> AND <RIGHT BRACKET> OR #CODE FOLLOWED       *  6500
  BY A NUMBER  OR A SHARP FOLLOWED BY ANY TERMINAL BUT NUMBER  OR        *  6600
  A <CHARACTER> OR AN IDENTIFIER OR A NUMBER OR #ALPHA FOLLOWED BY       *  6700
  AN <ALPHA CHARACTER STRING> OR #ALPHA FOLLOWED BY A LIST OF <ALPHA     *  6800
  CHARACTER STRING>S POSSIBLY SEPARATED BY COMMAS  ENCLOSED  IN          *  6900
  SQUARE BRACKETS]}                                                     *  7000
AN <ALPHA CHARACTER STRING> CONSISTS OF POSSIBLY ONE SHARP FOLLOWED     *  7100
  BY ANY NONCHARACTER}                                                  *  7200
AN <ENGLISH TERMINAL> IS                                                *  7300
    #IDENTIFIER                    OR #IDENTIFIERS                      *  7400
 OR #NUMBER                        OR #NUMBERS                          *  7500
 OR #STRING                        OR #STRINGS                          *  7600
 OR #COMMA                         OR #COMMAS                           *  7700
 OR #SEMICOLON                     OR #SEMICOLONS                       *  7800
 OR #PERIOD                        OR #PERIODS                          *  7900
 OR #SHARP                         OR #SHARPS                           *  8000
 OR #COLON                         OR #COLONS                           *  8100
 OR #LEFT #PARENTHESIS             OR #LEFT #PARENTHESES                *  8200
 OR #RIGHT #PARENTHESIS            OR #RIGHT #PARENTHESES               *  8300
 OR #LEFT #SQUARE #BRACKET         OR #LEFT #SQUARE #BRACKETS           *  8400
 OR #RIGHT #SQUARE #BRACKET        OR #RIGHT #SQUARE #BRACKETS          *  8500
 OR #LEFT #ANGLE #BRACKET          OR #LEFT #ANGLE #BRACKETS            *  8600
 OR #RIGHT #ANGLE #BRACKETS        OR #RIGHT #ANGLE #BRACKETS           *  8700
 OR #QUESTION #MARK                OR #QUESTION #MARKS                  *  8800
 OR #QUOTE                         OR #QUOTES                           *  8900
 OR #AMPERSAND                     OR #AMPERSANDS                       *  9000
 OR #ASTERISK                      OR #ASTERISKS                        *  9100
 OR #STAR                          OR #STARS                            *  9200
 OR #SLASH                         OR #SLASHES                          *  9300
```

```
 OR #VIRGULE                                OR #VIRGULES                 *
 OR #PERCENT #SIGN                          OR #PERCENT #SIGNS           *
 OR #DOLLAR #SIGN                           OR #DOLLAR #SIGNS            *
 OR #EQUAL #SIGN                            OR #EQUAL #SIGNS             *
 OR #NOT #EQUAL #SIGN                       OR #NOT #EQUAL #SIGNS        *
 OR #LESS #THAN #OR #EQUAL #TO [#SIGN       OR #SIGNS]                   *
 OR #GREATER #THAN #OR #EQUAL #TO [#SIGN OR #SIGNS]                      *
 OR #LESS #THAN #SIGN                       OR #LESS #THAN #SIGNS        *
 OR #GREATER #THAN #SIGN                    OR #GREATER #THAN #SIGNS     *
 OR #LEFT #ARROW                            OR #LEFT #ARROWS;            *
A <TERMINAL CLASS NAME> IS AN ASTERISK FOLLOWED BY EITHER [A NUMBER *
  OR AN #I  OR AN #N  OR AN #S];                                        *
A <CHARACTER> IS ANY CHARACTER BUT [ #/    OR   #<    OR   #>          *
                                  OR #"    OR   #(    OR   #)          *
                                  OR #:    OR   #;    OR   #@          *
                                  OR ##    OR   #&    OR   #*          *
                                  OR #+    OR   #[    OR   #] ];       *
A <LEFT RECURSIVE LIST> CONSISTS OF POSSIBLY ONE [#REDUCED OR #LEFT *
  #RECURSIVE] FOLLOWED BY A <BASIC LIST>  OR  A <BASIC LIST>           *
  FOLLOWED BY #CLOSE;                                                   *
A <RIGHT RECURSIVE LIST> CONSISTS OF [ #EXPANDING OR                    *
  #RIGHT #RECURSIVE  OR #R] FOLLOWED BY A <BASIC LIST> OR A <BASIC     *
  LIST> FOLLOWED BY #OPEN;                                              *
A <BASIC LIST> CONSISTS OF A <LIST TYPE> FOLLOWED BY A <SIMPLE         *
  SYNTACTIC SYMBOL> FOLLOWED BY POSSIBLY ONE #S FOLLOWED BY POSSIBLY*
  ONE [<SEPARATOR TYPE> FOLLOWED BY A <SIMPLE SYNTACTIC SYMBOL>        *
  FOLLOWED BY POSSIBLY ONE #S];                                         *
A <LIST TYPE> CONSISTS OF [                                             *
          #NONEMPTY #LIST                                               *
       OR #NONEMPTY #STRING                                             *
       OR #NONEMPTY #SEQUENCE                                           *
```

```
        OR #POSSIBLY #EMPTY #LIST                                    *  12500
        OR #POSSIBLY #EMPTY #STRING                                  *  12600
        OR #POSSIBLY #EMPTY #SEQUENCE                                *  12700
        OR #L                                                        *  12800
        OR #EL                                                       *  12900
        OR #LIST                                                     *  13000
        OR #KLEENE STAR                                              *  13100
        OR #SEQUENCE       ) FOLLOWED BY POSSIBLY ONE #OF            *  13200
   OR  (                                                             *  13300
           #STRING                                                   *  13400
        OR #KLEENE         ) FOLLOWED BY #OF)                        *  13500
A <SEPARATOR TYPE> IS EITHER (A (<DEFINITE SEPARATOR> WHICH IS       *  13600
   DEFINED TO BE #SEP  OR #SEPARATOR  OR #SEPARATED #BY)) OR (A      *  13700
   (<QUESTIONABLE SEPARATOR> WHICH IS #Q OR #POSSIBLY FOLLOWED BY A  *  13800
   <DEFINITE SEPARATOR>))                                            *  13900
A <SEMANTIC SYMBOL> IS DEFINED TO BE EITHER (A <TEST CALL> OR AN     *  14000
   <ACTION CALL> OR A <BIT TEST>  OR A <BIT ACTION>)                 *  14100
A <TEST CALL> CONSISTS OF #@ #T  FOLLOWED BY A <SEMANTIC ROUTINE     *  14200
   CALL>  OR (A SHARP FOLLOWED BY A <SEMANTIC ROUTINE CALL>) ENCLOSED*  14300
   IN <LEFT BRACKET> AND <RIGHT BRACKET>)                            *  14400
AN <ACTION CALL> CONSISTS OF A SHARP FOLLOWED BY A NUMBER OR EITHER  *  14500
   (#@ #S OR #@ #Q) FOLLOWED BY A <SEMANTIC ROUTINE CALL>)           *  14600
A <BIT TEST> CONSISTS OF #@ #T FOLLOWED BY #BIT  FOLLOWED BY ANY     *  14700
   NONCHARACTER FOLLOWED BY POSSIBLY ONE (#ON OR #OFF)               *  14800
   FOLLOWED BY POSSIBLY ONE (#SET OR #RESET))                        *  14900
A <BIT ACTION> CONSISTS OF #@ #S FOLLOWED BY EITHER (#SET OR #RESET)*  15000
   FOLLOWED BY #BIT FOLLOWED BY ANY NONCHARACTER)                    *  15100
A <SEMANTIC ROUTINE CALL> IS ANY NONCHARACTER BUT (#BEGIN OR #END)   *  15200
   FOLLOWED BY POSSIBLY ONE  LIST OF NUMBERS SEPARATED BY COMMAS     *  15300
   ENCLOSED IN PARENTHESES FOLLOWED BY POSSIBLY ONE (COLON FOLLOWED  *  15400
   BY A <CODE DECLARATION>)    OR  A  <CODE DECLARATION>)            *  15500
```

```
A <CODE DECLARATION> IS EITHER [<CODE> ENCLOSED IN SQUARE BRACKETS  *  15600
  OR <CODE> ENCLOSED IN #BEGIN AND #END];                           *  15700
<CODE> CONSISTS OF A POSSIBLY EMPTY LIST OF EITHER [ ANY  TERMINAL  *  15800
  BUT [#BEGIN OR #END OR #[ OR #]] OR <CODE DECLARATION>];          *  15900
AN <EMPTY SYMBOL> IS DEFINED TO BE #LAMBDA  OR #EMPTY OR A <LEFT    *  16000
  BRACKET> FOLLOWED BY A <RIGHT BRACKET>;                           *  16100
A <NULL SYMBOL> IS DEFINED TO BE #BACK <SIMPLE SYNTACTIC SYMBOL>    *  16200
  OR #AHEAD <SYMBOLIC TERMINAL>  OR #NOT <SYMBOLIC TERMINAL> OR     *  16300
  #NOBACK OR A[<COMMENT> WHICH CONSISTS OF  EITHER [#COMMENT OR     *  16400
  #C] FOLLOWED BY A POSSIBLY EMPTY LIST OF  ANY TERMINALS BUT PERIOD*  16500
  FOLLOWED BY A PERIOD];                                            *  16600
A <NONTERMINAL> IS A LIST OF ANY TERMINALS BUT [#> OR #"] STARTING  *  16700
  WITH ANY TERMINAL BUT [SHARP OR STAR] ENCLOSED IN <LEFT BRACKET>  *  16800
  AND <RIGHT BRACKET>;                                              *  16900
A <LEFT BRACKET> IS EITHER [#< OR #"];                              *  17000
A <RIGHT BRACKET> IS EITHER [#> OR #"];                             *  17100
A <SEMANTIC TAIL> IS A <CODE DECLARATION>;                          *  17200
A <LANGUAGE NAME DESIGNATION> CONSISTS OF #LANGUAGE FOLLOWED BY     *  17300
  COLON FOLLOWED BY ANY NONCHARACTER;                               *  17400
A <CONTROL STATEMENT> IS  EITHER [                                  *  17500
     #PRINT #: FOLLOWED BY A LIST OF <PRINT OPTION>S SEPARATED BY   *  17600
          COMMAS                                                    *  17700
  OR #CLOSE FOLLOWED BY EITHER [#LP OR #LINEPRINTER                 *  17800
                                  OR #LINE #PRINTER OR EMPTY]       *  17900
  OR #COMBINED #FIRST                                               *  18000
  OR #LONG #LOOKAHEAD                                               *  18100
  OR <PROGRAM NAME> FOLLOWED BY [ #PRIORITY  #=  NUMBER OR          *  18200
                                  #CORE  #=  NUMBER  OR             *  18300
                                  #STACK  #=  NUMBER]               *  18400
  OR [#RECURSIVE #DESCENT  OR  #FLOYD #PRODUCTION]  #PARSER         *  18500
  OR #EXECUTABLE  [ #FLOYD #PRODUCTIONS OR                          *  18600
```

```
                    #LOOKAHEAD  OR                                          *  18700
                    #FILL #TABLES )                                         *  18800
    OR #FLOYD #PRODUCTIONS #PER #PROCEDURE #: NUMBER                        *  18900
    OR #GROUPS #PER #PROCEDURE  #:  NUMBER                                  *  19000
    OR #PROGRAM  #SYMBOL  #: FOLLOWED BY ANY NONCHARACTER BUT NUMBER        *  19100
    OR #SPECIAL #SYMBOLS #: FOLLOWED BY A LIST OF ANY NONCHARACTERS         *  19200
          BUT NUMBER  SEPARATED BY COMMAS)!                                 *  19300
  A <PROGRAM NAME>  IS   #TWST      OR #COMPILER  OR                        *  19400
                         #ALGOL     OR #ISL       OR                        *  19500
                         #BNF2FPL   OR #FPL2PAR   OR #PAR2ALG!            *  19600
  A <PRINT OPTION>  IS                                                      *  19700
          #TABLESF #SIZES      OR  #CHARACTERS                              *  19800
       OR #TERMINALS           OR  #TERMINALS  #ALPHABETICALLY              *  19900
       OR #TRMCHR              OR  #TERMINALS  #NUMERICALLY                 *  20000
       OR #TRMALF              OR  #TRMNUM                                  *  20100
       OR #NONTERMINALS        OR  #NONTERMINALS #ALPHABETICALLY            *  20200
       OR #NTALF               OR  #NONTERMINALS #NUMERICALLY               *  20300
       OR #NTNUM               OR  #SYNTAX                                  *  20400
       OR #INPUT               OR  #CROSS #REFERENCE                        *  20500
       OR #XREF                OR  #INDEX                                   *  20600
       OR #PATTERNS            OR  #ACTIONS                                 *  20700
       OR #ACTALF              OR  #ACTIONS #ALPHABETICALLY                 *  20800
       OR #ACTNUM              OR  #ACTIONS #NUMERICALLY                    *  20900
       OR #PROTAB              OR #FLOYD                                    *  21000
       OR #COMBINED #GROUPS    OR #PARSER                                   *  21100
       OR #STANDARD            OR #DEBUGGING                                *  21200
       OR #DEBUGN              OR #EVERYTHING                               *  21300
       OR #NOTHING!                                                         *  21400

        3 MINUTES 13.3 SECONDS FOR THIS GROUP!          3 MINUTES 13.3 SECONDS TOT
END OF INITIAL SCANNING.
```

COMPLETE PROTAB IS:

```
      LOC.  LEFT HAND SIDE                    RIGHT HAND SIDE

       1: <WHOLE PROGRAM                  > ::= # EOF MARK
       2:                                       <TWINKLEPROGRAM                  >
       3:                                       # EOF MARK
       4: <TWINKLEPROGRAM                 > ::= <LANGUAGESYNTACTI DUMMY 2       >
       5:                                       <DUMMY 73                        >
       6: <TWINKLEPROGRAM                 > ::= <LANGUAGESYNTACTI DUMMY 2       >
       7:                                       <DUMMY 75                        >
       8: <TWINKLEPROGRAM                 > ::= <LANGUAGESYNTACTI DUMMY 2       >
       9:                                       #END
      10: <TWINKLEPROGRAM                 > ::= <LANGUAGESYNTACTI DUMMY 2       >
      11:                                       #END
      12:                                       <CODEDECLARATION                 >
      13: <TWINKLEPROGRAM                 > ::= <LANGUAGESYNTACTI DUMMY 2       >
      14: <TWINKLEPROGRAM                 > ::= <LANGUAGESYNTACTI DUMMY 2       >
      15:                                       <CODEDECLARATION                 >
      16: <CONTROLSTATEMENT               > ::= #CLOSE
      17:                                       #LP
      18: <CONTROLSTATEMENT               > ::= #CLOSE
      19:                                       #LINEPRINTER
      20: <CONTROLSTATEMENT               > ::= #CLOSE
      21:                                       #LINE
      22:                                       #PRINTER
      23: <CONTROLSTATEMENT               > ::= #CLOSE
      24: <CONTROLSTATEMENT               > ::= #COMBINED
      25:                                       #FIRST
      26: <CONTROLSTATEMENT               > ::= #LONG
      27:                                       #LOOKAHEAD
      28: <CONTROLSTATEMENT               > ::= <PROGRAMNAME                   >
      29:                                       #PRIORITY
      30:                                        "="
      31:                                        NUMERIC LITERAL
      32: <CONTROLSTATEMENT               > ::= <PROGRAMNAME                   >
      33:                                       #CORE
      34:                                        "="
      35:                                        NUMERIC LITERAL
      36: <CONTROLSTATEMENT               > ::= <PROGRAMNAME                   >
      37:                                       #STACK
      38:                                        "="
      39:                                        NUMERIC LITERAL
      40: <CONTROLSTATEMENT               > ::= #RECURSIVE
      41:                                       #DESCENT
      42:                                       #PARSER
      43: <CONTROLSTATEMENT               > ::= #FLOYD
      44:                                       #PRODUCTION
      45:                                       #PARSER
      46: <CONTROLSTATEMENT               > ::= #EXECUTABLE
      47:                                       #FLOYD
      48:                                       #PRODUCTIONS
      49: <CONTROLSTATEMENT               > ::= #EXECUTABLE
      50:                                       #LOOKAHEAD
      51: <CONTROLSTATEMENT               > ::= #EXECUTABLE
      52:                                       #FILL
      53:                                       #TABLES
      54: <CONTROLSTATEMENT               > ::= #FLOYD
```

```
                                        #PRODUCTIONS
                                        #PER
                                        #PROCEDURE
                                         ";"
                                         NUMERIC LITERAL
<CONTROLSTATEMENT                > ::= #GROUPS
                                        #PER
                                        #PROCEDURE
                                         ";"
                                         NUMERIC LITERAL
<CONTROLSTATEMENT                > ::= #PROGRAM
                                        #SYMBOL
                                         ";"
                                         ANY SYMBOL    13
<CONTROLSTATEMENT                > ::= <CONTROLSTATEMENT DUMMY 20      >
<CONTROLSTATEMENT                > ::= <CONTROLSTATEMENT DUMMY 21      >
<LANGUAGESYNTACTI DUMMY 1        > ::= <LANGUAGESYNTACTI DUMMY 1       >
                                         ";"
                                        <CONTROLSTATEMENT                >
<LANGUAGESYNTACTI DUMMY 1        > ::= #LANGUAGE
                                         ":"
                                         ANY SYMBOL     5
                                         ";"
                                        <CONTROLSTATEMENT                >
<LANGUAGESYNTACTI DUMMY 2        > ::= <LANGUAGESYNTACTI DUMMY 2       >
                                        <DUMMY 74                        >
<LANGUAGESYNTACTI DUMMY 2        > ::= #LANGUAGE
                                         ":"
                                         ANY SYMBOL     5
                                         ";"
                                        <PRODUCTION DUMMY 4              >
<LANGUAGESYNTACTI DUMMY 2        > ::= <LANGUAGESYNTACTI DUMMY 1       >
                                         ";"
                                        <PRODUCTION DUMMY 4              >
<PRODUCTION DUMMY 3              > ::= #A
<PRODUCTION DUMMY 3              > ::= #AN
<NONTERMINAL                     > ::= <LEFTBRACKET                       >
                                        <DUMMY 76                        >
<NONTERMINAL                     > ::= <NONTERMINAL DUMMY 19              >
                                        <RIGHTBRACKET                    >
<ARROW                           > ::=  ":"
                                         ":"
                                         "="
<ARROW                           > ::= #IS
<ARROW                           > ::= #IS
                                        #DEFINED
                                        #TO
                                        #BE
<ARROW                           > ::= #CONSISTS
                                        #OF
<DEFINITION                      > ::= <PRODUCTION DUMMY 3             >
                                        <DUMMY 103                       >
<DEFINITION                      > ::= <DEFINITION DUMMY 5             >
<DEFINITION                      > ::= <SYMBOL                         >
<PRODUCTION DUMMY 4              > ::= <PRODUCTION DUMMY 4             >
                                         "/"
                                        <DEFINITION                      >
<PRODUCTION DUMMY 4              > ::= <PRODUCTION DUMMY 4             >
                                        #OR
```

```
114:                                                <DEFINITION                                  >
115: <PRODUCTION DUMMY 4                    > ::= <PRODUCTION DUMMY 3                          >
116:                                                <DUMMY 79                                    >
117: <PRODUCTION DUMMY 4                    > ::= <NONTERMINAL                                 >
118:                                                <ARROW                                       >
119:                                                <DEFINITION                                  >
120: <SYMBOL                                > ::= <SYNTACTICSYMBOL                             >
121: <SYMBOL                                > ::= <SEMANTICSYMBOL                              >
122: <SYMBOL                                > ::= <NULLSYMBOL                                  >
123: <SYMBOL                                > ::= <EMPTYSYMBOL                                 >
124: <DEFINITION DUMMY 5                    > ::= <DEFINITION DUMMY 5                          >
125:                                                <DUMMY 82                                    >
126: <DEFINITION DUMMY 5                    > ::= <DEFINITION DUMMY 5                          >
127:                                                <SYMBOL                                      >
128: <DEFINITION DUMMY 5                    > ::= <PRODUCTION DUMMY 3                          >
129:                                                <DUMMY 105                                   >
130: <DEFINITION DUMMY 5                    > ::= <PRODUCTION DUMMY 3                          >
131:                                                <DUMMY 106                                   >
132: <DEFINITION DUMMY 5                    > ::= <DEFINITION DUMMY 5                          >
133:                                                <DUMMY 83                                    >
134: <DEFINITION DUMMY 5                    > ::= <SYMBOL                                      >
135:                                                <DUMMY 86                                    >
136: <DEFINITION DUMMY 5                    > ::= <PRODUCTION DUMMY 3                          >
137:                                                <DUMMY 107                                   >
138: <DEFINITION DUMMY 5                    > ::= <SYMBOL                                      >
139:                                                <SYMBOL                                      >
140: <DEFINITION DUMMY 5                    > ::= <SYMBOL                                      >
141:                                                <DUMMY 87                                    >
142: <SYNTACTICSYMBOL                       > ::= <SIMPLESYNTACTICSYMBOL                       >
143:                                                "*"
144: <SYNTACTICSYMBOL                       > ::= <SIMPLESYNTACTICSYMBOL                       >
145:                                                "?"
146: <SYNTACTICSYMBOL                       > ::= #POSSIBLY
147:                                                #ONE
148:                                                <GENERALSYNTACTICSYMBOL                      >
149: <SYNTACTICSYMBOL                       > ::= <GENERALSYNTACTICSYMBOL                      >
150: <SEMANTICSYMBOL                        > ::= <TESTCALL                                    >
151: <SEMANTICSYMBOL                        > ::= <ACTIONCALL                                  >
152: <SEMANTICSYMBOL                        > ::= <BITTEST                                     >
153: <SEMANTICSYMBOL                        > ::= <BITACTION                                   >
154: <NULLSYMBOL                            > ::= #BACK
155:                                                <SIMPLESYNTACTICSYMBOL                       >
156: <NULLSYMBOL                            > ::= #AHEAD
157:                                                <SYMBOLICTERMINAL                            >
158: <NULLSYMBOL                            > ::= #NOT
159:                                                <SYMBOLICTERMINAL                            >
160: <NULLSYMBOL                            > ::= #NOBACK
161: <NULLSYMBOL                            > ::= <NULLSYMBOL DUMMY 17                         >
162: <EMPTYSYMBOL                           > ::= #LAMBDA
163: <EMPTYSYMBOL                           > ::= #EMPTY
164: <EMPTYSYMBOL                           > ::= <LEFTBRACKET                                 >
165:                                                <RIGHTBRACKET                                >
166: <SIMPLESYNTACTICSYMBOL                 > ::= <TERMINAL                                    >
167: <SIMPLESYNTACTICSYMBOL                 > ::= <ANYSYMBOL                                   >
168: <SIMPLESYNTACTICSYMBOL                 > ::= <NONTERMINAL                                 >
169: <SIMPLESYNTACTICSYMBOL                 > ::= <SQUAREBRACKETCON DUMMY 7                    >
170: <SIMPLESYNTACTICSYMBOL                 > ::= <SQUAREBRACKETCON DUMMY 7                    >
171:                                                <KLEENESTAR                                  >
172: <SIMPLESYNTACTICSYMBOL                 > ::= <NESTEDPRODUCTION DUMMY 8                    >
```

```
173: <SIMPLESYNTACTICSYMBOL          > ::= <NESTEDPRODUCTION DUMMY 8      >
174:                                       <KLEENESTAR                    >
175: <SIMPLESYNTACTICSYMBOL          > ::= #EITHER
176:                                       <SQUAREBRACKETCON DUMMY 7      >
177: <SIMPLESYNTACTICSYMBOL          > ::= #EITHER
178:                                       <SQUAREBRACKETCON DUMMY 7      >
179:                                       <KLEENESTAR                    >
180: <SIMPLESYNTACTICSYMBOL          > ::= #EITHER
181:                                       <DUMMY 90                      >
182: <SIMPLESYNTACTICSYMBOL          > ::= #EITHER
183:                                       <DUMMY 91                      >
184: <GENERALSYNTACTICSYMBOL         > ::= <SEEDEDLIST                    >
185: <GENERALSYNTACTICSYMBOL         > ::= <SIMPLESYNTACTICSYMBOL         >
186: <GENERALSYNTACTICSYMBOL         > ::= <GENERALSYNTACTIC DUMMY 6      >
187: <SEEDEDLIST                     > ::= <LEFTRECURSIVELIST             >
188: <SEEDEDLIST                     > ::= <LEFTRECURSIVELIST             >
189:                                       <SEEDEDLIST DUMMY 9            >
190: <SEEDEDLIST                     > ::= <RIGHTRECURSIVELIST            >
191: <SEEDEDLIST                     > ::= <RIGHTRECURSIVELIST            >
192:                                       <SEEDEDLIST DUMMY 9            >
193: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
194:                                       #IN
195:                                       <DUMMY 92                      >
196: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
197:                                       #IN
198:                                       <DUMMY 93                      >
199: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
200:                                       #IN
201:                                       <DUMMY 94                      >
202: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
203:                                       #IN
204:                                       <SIMPLESYNTACTICSYMBOL         >
205:                                       #AND
206:                                       <SIMPLESYNTACTICSYMBOL         >
207: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
208:                                       #IN
209:                                       <SIMPLESYNTACTICSYMBOL         >
210:                                       #AND
211:                                       <SIMPLESYNTACTICSYMBOL         >
212:                                       #S
213: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
214:                                       #IN
215:                                       <SIMPLESYNTACTICSYMBOL         >
216: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
217:                                       #IN
218:                                       <SIMPLESYNTACTICSYMBOL         >
219:                                       #S
220:                                       #AND
221:                                       <SIMPLESYNTACTICSYMBOL         >
222: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
223:                                       #IN
224:                                       <SIMPLESYNTACTICSYMBOL         >
225:                                       #S
226:                                       #AND
227:                                       <SIMPLESYNTACTICSYMBOL         >
228:                                       #S
229: <ENCLOSUREOPERATOR              > ::= #ENCLOSED
230:                                       #IN
231:                                       <SIMPLESYNTACTICSYMBOL         >
```

```
                                        #S
<GENERALSYNTACTIC DUMMY 6        > ::= <GENERALSYNTACTIC DUMMY 6        >
                                        <ENCLOSUREOPERATOR               >
<GENERALSYNTACTIC DUMMY 6        > ::= <SEEDEDLIST                      >
                                        <ENCLOSUREOPERATOR               >
<GENERALSYNTACTIC DUMMY 6        > ::= <SIMPLESYNTACTICSYMBOL           >
                                        <ENCLOSUREOPERATOR               >
<TERMINAL                        > ::= <SYMBOLICTERMINAL                >
<TERMINAL                        > ::= <ENGLISHTERMINAL                 >
<ANYSYMBOL                       > ::= <ANYBASE                         >
<ANYSYMBOL                       > ::= <ANYBASE                         >
                                        <EXCEPTION                       >
<KLEENESTAR                      > ::= #CLOSE
                                        "*"
<KLEENESTAR                      > ::= #CLOSE
                                        "+"
<KLEENESTAR                      > ::= #OPEN
                                        "*"
<KLEENESTAR                      > ::= #OPEN
                                        "+"
<KLEENESTAR                      > ::=  "*"
<KLEENESTAR                      > ::=  "+"
<SQUAREBRACKETCON DUMMY 7        > ::= <SQUAREBRACKETCON DUMMY 7        >
                                        "/"
                                        <DEFINITION                      >
<SQUAREBRACKETCON DUMMY 7        > ::= <SQUAREBRACKETCON DUMMY 7        >
                                        #OR
                                        <DEFINITION                      >
<SQUAREBRACKETCON DUMMY 7        > ::=  "["
                                        <DEFINITION                      >
<SQUAREBRACKETCON DUMMY 7        > ::= <DEFINITION                      >
<NESTEDPRODUCTION DUMMY 8        > ::= <NESTEDPRODUCTION DUMMY 8        >
                                        "/"
                                        <DEFINITION                      >
<NESTEDPRODUCTION DUMMY 8        > ::= <NESTEDPRODUCTION DUMMY 8        >
                                        #OR
                                        <DEFINITION                      >
<NESTEDPRODUCTION DUMMY 8        > ::=  "["
                                        <DUMMY 95                        >
<NESTEDPRODUCTION DUMMY 8        > ::=  "["
                                        <DUMMY 97                        >
<NESTEDPRODUCTION DUMMY 8        > ::= <PRODUCTION DUMMY 3              >
                                        <DUMMY 80                        >
<NESTEDPRODUCTION DUMMY 8        > ::= <PRODUCTION DUMMY 3              >
                                        <DUMMY 81                        >
<NESTEDPRODUCTION DUMMY 8        > ::=  "["
                                        <DUMMY 99                        >
<NESTEDPRODUCTION DUMMY 8        > ::=  "["
                                        <DUMMY 100                       >
<NESTEDPRODUCTION DUMMY 8        > ::= <NONTERMINAL                     >
                                        <ARROW                           >
                                        <DEFINITION                      >
<NESTEDPRODUCTION DUMMY 8        > ::= <NONTERMINAL                     >
                                        #WHICH
                                        <ARROW                           >
                                        <DEFINITION                      >
<LEFTRECURSIVELIST               > ::= #REDUCED
                                        <BASICLIST                       >
<LEFTRECURSIVELIST               > ::= #LEFT
```

```
291:
292:
293: <LEFTRECURSIVELIST                > ::= <BASICLIST                    >
294: <LEFTRECURSIVELIST                > ::= <BASICLIST                    >
                                             #CLOSE
295:
296: <RIGHTRECURSIVELIST               > ::= #EXPANDING
                                             <BASICLIST                    >
297:
298: <RIGHTRECURSIVELIST               > ::= #RIGHT
                                             #RECURSIVE
299:                                         <BASICLIST                    >
300:
301: <RIGHTRECURSIVELIST               > ::= #R
                                             <BASICLIST                    >
302:
303: <RIGHTRECURSIVELIST               > ::= <BASICLIST                    >
                                             #OPEN
304:
305: <SEEDEDLIST DUMMY 9               > ::= #STARTING
                                             #WITH
306:                                         <DUMMY 102                    >
307:
308: <SEEDEDLIST DUMMY 9               > ::= #STARTING
                                             #WITH
309:                                         <DUMMY 108                    >
310:
311: <SEEDEDLIST DUMMY 9               > ::= #BEGINNING
                                             #WITH
312:                                         <DUMMY 110                    >
313:
314: <SEEDEDLIST DUMMY 9               > ::= #BEGINNING
                                             #WITH
315:                                         <DUMMY 112                    >
316:
317: <SEEDEDLIST DUMMY 9               > ::= #STARTING
                                             #WITH
318:                                         <SIMPLESYNTACTICSYMBOL        >
319:
320: <SEEDEDLIST DUMMY 9               > ::= #STARTING
                                             #WITH
321:                                         <DUMMY 113                    >
322:
323: <SEEDEDLIST DUMMY 9               > ::= #BEGINNING
                                             #WITH
324:                                         <SIMPLESYNTACTICSYMBOL        >
325:
326: <SEEDEDLIST DUMMY 9               > ::= #BEGINNING
                                             #WITH
327:                                         <DUMMY 114                    >
328:
329: <ANYBASE                          > ::= #ANY
                                             <NONTERMINAL                  >
330:
331: <ANYBASE                          > ::= #ANY
                                             <DUMMY 115                    >
332:
333: <ANYBASE                          > ::= #ANY
                                             <DUMMY 116                    >
334:
335: <ANYBASE                          > ::= #ANY
                                             <DUMMY 117                    >
336:
337: <ANYBASE                          > ::= #ANY
                                             <DUMMY 118                    >
338:
339: <ANYBASE                          > ::= #ANY
                                             <DUMMY 119                    >
340:
341: <ANYBASE                          > ::= #ANY
                                             <DUMMY 120                    >
342:
343: <ANYBASE                          > ::= #ANY
                                             <DUMMY 121                    >
344:
345: <ANYBASE                          > ::= #ANY
                                             <DUMMY 122                    >
346:
347: <ANYBASE                          > ::= #ANY
                                             <DUMMY 123                    >
348:
349: <ANYBASE                          > ::= #ANY
```

```
350:                                                <DUMMY 124                                  >
351: <ANYBASE                                  > ::= $ANY
352:                                                <DUMMY 125                                  >
353: <ANYBASE                                  > ::= $ANY
354:                                                <DUMMY 126                                  >
355: <ANYBASE                                  > ::= $ANY
356:                                                <DUMMY 127                                  >
357: <ANYBASE                                  > ::= $ANY
358:                                                <DUMMY 128                                  >
359: <ANYBASE                                  > ::= $ANY
360:                                                <DUMMY 129                                  >
361: <ANYBASE                                  > ::= $ANY
362:                                                <DUMMY 130                                  >
363: <EXCEPTION                                > ::= <EXCEPTION DUMMY 10                       >
364: <EXCEPTION                                > ::= <EXCEPTION DUMMY 11                       >
365: <EXCEPTION DUMMY 10                       > ::= <EXCEPTION DUMMY 10                       >
366:                                                $BUT
367:                                                <TERMINAL                                   >
368: <EXCEPTION DUMMY 10                       > ::= $BUT
369:                                                <TERMINAL                                   >
370: <EXCEPTION DUMMY 11                       > ::= <EXCEPTION DUMMY 11                       >
371:                                                 "/"
372:                                                <TERMINAL                                   >
373: <EXCEPTION DUMMY 11                       > ::= <EXCEPTION DUMMY 11                       >
374:                                                <DUMMY 133                                  >
375: <EXCEPTION DUMMY 11                       > ::= <EXCEPTION DUMMY 11                       >
376:                                                <TERMINAL                                   >
377: <EXCEPTION DUMMY 11                       > ::= $BUT
378:                                                <DUMMY 131                                  >
379: <EXCEPTION DUMMY 11                       > ::= $BUT
380:                                                <TERMINAL                                   >
381: <SYMBOLICTERMINAL                         > ::= $BLANK
382: <SYMBOLICTERMINAL                         > ::= <LEFTBRACKET                              >
383:                                                <TERMINALCLASSNAME                          >
384:                                                <RIGHTBRACKET                               >
385: <SYMBOLICTERMINAL                         > ::= $CODE
386:                                                 NUMERIC LITERAL
387: <SYMBOLICTERMINAL                         > ::=  "$"
388:                                                 ANY SYMBOL     6
389: <SYMBOLICTERMINAL                         > ::=  ANY SYMBOL     7
390: <SYMBOLICTERMINAL                         > ::=  IDENTIFIER
391: <SYMBOLICTERMINAL                         > ::=  NUMERIC LITERAL
392: <SYMBOLICTERMINAL                         > ::= $ALPHA
393:                                                <ALPHACHARACTERSTRING                       >
394: <SYMBOLICTERMINAL                         > ::= <SYMBOLICTERMINAL DUMMY 12                >
395: <ENGLISHTERMINAL                          > ::= $IDENTIFIER
396: <ENGLISHTERMINAL                          > ::= $IDENTIFIERS
397: <ENGLISHTERMINAL                          > ::= $NUMBER
398: <ENGLISHTERMINAL                          > ::= $NUMBERS
399: <ENGLISHTERMINAL                          > ::= $STRING
400: <ENGLISHTERMINAL                          > ::= $STRINGS
401: <ENGLISHTERMINAL                          > ::= $COMMA
402: <ENGLISHTERMINAL                          > ::= $COMMAS
403: <ENGLISHTERMINAL                          > ::= $SEMICOLON
404: <ENGLISHTERMINAL                          > ::= $SEMICOLONS
405: <ENGLISHTERMINAL                          > ::= $PERIOD
406: <ENGLISHTERMINAL                          > ::= $PERIODS
407: <ENGLISHTERMINAL                          > ::= $SHARP
408: <ENGLISHTERMINAL                          > ::= $SHARPS
```

```
409: <ENGLISHTERMINAL                    > ::= #COLON
410: <ENGLISHTERMINAL                    > ::= #COLONS
411: <ENGLISHTERMINAL                    > ::= #LEFT
412:                                           #PARENTHESIS
413: <ENGLISHTERMINAL                    > ::= #LEFT
414:                                           #PARENTHESES
415: <ENGLISHTERMINAL                    > ::= #RIGHT
416:                                           #PARENTHESIS
417: <ENGLISHTERMINAL                    > ::= #RIGHT
418:                                           #PARENTHESES
419: <ENGLISHTERMINAL                    > ::= #LEFT
420:                                           #SQUARE
421:                                           #BRACKET
422: <ENGLISHTERMINAL                    > ::= #LEFT
423:                                           #SQUARE
424:                                           #BRACKETS
425: <ENGLISHTERMINAL                    > ::= #RIGHT
426:                                           #SQUARE
427:                                           #BRACKET
428: <ENGLISHTERMINAL                    > ::= #RIGHT
429:                                           #SQUARE
430:                                           #BRACKETS
431: <ENGLISHTERMINAL                    > ::= #LEFT
432:                                           #ANGLE
433:                                           #BRACKET
434: <ENGLISHTERMINAL                    > ::= #LEFT
435:                                           #ANGLE
436:                                           #BRACKETS
437: <ENGLISHTERMINAL                    > ::= #RIGHT
438:                                           #ANGLE
439:                                           #BRACKETS
440: <ENGLISHTERMINAL                    > ::= #QUESTION
441:                                           #MARK
442: <ENGLISHTERMINAL                    > ::= #QUESTION
443:                                           #MARKS
444: <ENGLISHTERMINAL                    > ::= #QUOTE
445: <ENGLISHTERMINAL                    > ::= #QUOTES
446: <ENGLISHTERMINAL                    > ::= #AMPERSAND
447: <ENGLISHTERMINAL                    > ::= #AMPERSANDS
448: <ENGLISHTERMINAL                    > ::= #ASTERISK
449: <ENGLISHTERMINAL                    > ::= #ASTERISKS
450: <ENGLISHTERMINAL                    > ::= #STAR
451: <ENGLISHTERMINAL                    > ::= #STARS
452: <ENGLISHTERMINAL                    > ::= #SLASH
453: <ENGLISHTERMINAL                    > ::= #SLASHES
454: <ENGLISHTERMINAL                    > ::= #VIRGULE
455: <ENGLISHTERMINAL                    > ::= #VIRGULES
456: <ENGLISHTERMINAL                    > ::= #PERCENT
457:                                           #SIGN
458: <ENGLISHTERMINAL                    > ::= #PERCENT
459:                                           #SIGNS
460: <ENGLISHTERMINAL                    > ::= #DOLLAR
461:                                           #SIGN
462: <ENGLISHTERMINAL                    > ::= #DOLLAR
463:                                           #SIGNS
464: <ENGLISHTERMINAL                    > ::= #EQUAL
465:                                           #SIGN
466: <ENGLISHTERMINAL                    > ::= #EQUAL
467:                                           #SIGNS
```

```
468: <ENGLISHTERMINAL                > ::= #NOT
469:                                       <DUMMY 88                    >
470: <ENGLISHTERMINAL                > ::= #NOT
471:                                       <DUMMY 89                    >
472: <ENGLISHTERMINAL                > ::= #LESS
473:                                       #THAN
474:                                       #OR
475:                                       #EQUAL
476:                                       #TO
477:                                       #SIGN
478: <ENGLISHTERMINAL                > ::= #LESS
479:                                       #THAN
480:                                       #OR
481:                                       #EQUAL
482:                                       #TO
483:                                       #SIGNS
484: <ENGLISHTERMINAL                > ::= #GREATER
485:                                       #THAN
486:                                       #OR
487:                                       #EQUAL
488:                                       #TO
489:                                       #SIGN
490: <ENGLISHTERMINAL                > ::= #GREATER
491:                                       #THAN
492:                                       #OR
493:                                       #EQUAL
494:                                       #TO
495:                                       #SIGNS
496: <ENGLISHTERMINAL                > ::= #LESS
497:                                       #THAN
498:                                       #SIGN
499: <ENGLISHTERMINAL                > ::= #LESS
500:                                       #THAN
501:                                       #SIGNS
502: <ENGLISHTERMINAL                > ::= #GREATER
503:                                       #THAN
504:                                       #SIGN
505: <ENGLISHTERMINAL                > ::= #GREATER
506:                                       #THAN
507:                                       #SIGNS
508: <ENGLISHTERMINAL                > ::= #LEFT
509:                                       #ARROW
510: <ENGLISHTERMINAL                > ::= #LEFT
511:                                       #ARROWS
512: <TERMINALCLASSNAME              > ::=  "*"
513:                                        NUMERIC LITERAL
514: <TERMINALCLASSNAME              > ::=  "*"
515:                                       #I
516: <TERMINALCLASSNAME              > ::=  "*"
517:                                       #N
518: <TERMINALCLASSNAME              > ::=  "*"
519:                                       #S
520: <LEFTBRACKET                    > ::=  "<"
521: <LEFTBRACKET                    > ::=  """
522: <RIGHTBRACKET                   > ::=  ">"
523: <RIGHTBRACKET                   > ::=  """
524: <ALPHACHARACTERSTRING           > ::=  ANY SYMBOL     5
525: <ALPHACHARACTERSTRING           > ::=  "#"
526:                                        ANY SYMBOL     5
```

```
527: <SYMBOLICTERMINAL DUMMY 12      > ::= <SYMBOLICTERMINAL DUMMY 12      >
528:                                       ","
529:                                       <ALPHACHARACTERSTRING          >
530: <SYMBOLICTERMINAL DUMMY 12      > ::= <SYMBOLICTERMINAL DUMMY 12      >
531:                                       <ALPHACHARACTERSTRING          >
532: <SYMBOLICTERMINAL DUMMY 12      > ::= #ALPHA
533:                                       <ALPHACHARACTERSTRING          >
534: <SYMBOLICTERMINAL DUMMY 12      > ::= "["
535:                                       <ALPHACHARACTERSTRING          >
536: <BASICLIST                      > ::= <LISTTYPE                       >
537:                                       <SIMPLESYNTACTICSYMBOL         >
538:                                       <SEPARATORTYPE                 >
539:                                       <SIMPLESYNTACTICSYMBOL         >
540: <BASICLIST                      > ::= <LISTTYPE                       >
541:                                       <SIMPLESYNTACTICSYMBOL         >
542:                                       <SEPARATORTYPE                 >
543:                                       <SIMPLESYNTACTICSYMBOL         >
544:                                       #S
545: <BASICLIST                      > ::= <LISTTYPE                       >
546:                                       <SIMPLESYNTACTICSYMBOL         >
547: <BASICLIST                      > ::= <LISTTYPE                       >
548:                                       <SIMPLESYNTACTICSYMBOL         >
549:                                       #S
550:                                       <SEPARATORTYPE                 >
551:                                       <SIMPLESYNTACTICSYMBOL         >
552: <BASICLIST                      > ::= <LISTTYPE                       >
553:                                       <SIMPLESYNTACTICSYMBOL         >
554:                                       #S
555:                                       <SEPARATORTYPE                 >
556:                                       <SIMPLESYNTACTICSYMBOL         >
557:                                       #S
558: <BASICLIST                      > ::= <LISTTYPE                       >
559:                                       <SIMPLESYNTACTICSYMBOL         >
560:                                       #S
561: <LISTTYPE                       > ::= #NONEMPTY
562:                                       #LIST
563: <LISTTYPE                       > ::= #NONEMPTY
564:                                       #LIST
565:                                       #OF
566: <LISTTYPE                       > ::= #NONEMPTY
567:                                       #STRING
568: <LISTTYPE                       > ::= #NONEMPTY
569:                                       #STRING
570:                                       #OF
571: <LISTTYPE                       > ::= #NONEMPTY
572:                                       #SEQUENCE
573: <LISTTYPE                       > ::= #NONEMPTY
574:                                       #SEQUENCE
575:                                       #OF
576: <LISTTYPE                       > ::= #POSSIBLY
577:                                       #EMPTY
578:                                       #LIST
579: <LISTTYPE                       > ::= #POSSIBLY
580:                                       #EMPTY
581:                                       #LIST
582:                                       #OF
583: <LISTTYPE                       > ::= #POSSIBLY
584:                                       #EMPTY
585:                                       #STRING
```

```
586: <LISTTYPE                          > ::= #POSSIBLY
587:                                          #EMPTY
588:                                          #STRING
589:                                          #OF
590: <LISTTYPE                          > ::= #POSSIBLY
591:                                          #EMPTY
592:                                          #SEQUENCE
593: <LISTTYPE                          > ::= #POSSIBLY
594:                                          #EMPTY
595:                                          #SEQUENCE
596:                                          #OF
597: <LISTTYPE                          > ::= #L
598: <LISTTYPE                          > ::= #L
599:                                          #OF
600: <LISTTYPE                          > ::= #EL
601: <LISTTYPE                          > ::= #FL
602:                                          #OF
603: <LISTTYPE                          > ::= #LIST
604: <LISTTYPE                          > ::= #LIST
605:                                          #OF
606: <LISTTYPE                          > ::= #KLEENE
607:                                           "*"
608: <LISTTYPE                          > ::= #KLEENE
609:                                           "*"
610:                                          #OF
611: <LISTTYPE                          > ::= #SEQUENCE
612: <LISTTYPE                          > ::= #SEQUENCE
613:                                          #OF
614: <LISTTYPE                          > ::= #STRING
615:                                          #OF
616: <LISTTYPE                          > ::= #KLEENE
617:                                          #OF
618: <SEPARATORTYPE                     > ::= <SEPARATORTYPE DUMMY 13        >
619: <SEPARATORTYPE                     > ::= <SEPARATORTYPE DUMMY 14        >
620: <SEPARATORTYPE DUMMY 13            > ::= #SEP
621: <SEPARATORTYPE DUMMY 13            > ::= #SEPARATOR
622: <SEPARATORTYPE DUMMY 13            > ::= #SEPARATED
623:                                          #BY
624: <SEPARATORTYPE DUMMY 14            > ::= #O
625: <SEPARATORTYPE DUMMY 14            > ::= #POSSIBLY
626:                                          <SEPARATORTYPE DUMMY 13        >
627: <TESTCALL                          > ::=  "@"
628:                                          #T
629:                                          <SEMANTICROUTINECALL           >
630: <TESTCALL                          > ::= <LEFTBRACKET                   >
631:                                           "#"
632:                                          <SEMANTICROUTINECALL           >
633:                                          <RIGHTBRACKET                  >
634: <TESTCALL                          > ::=  "@"
635:                                          #T
636: <TESTCALL                          > ::= <LEFTBRACKET                   >
637:                                           "#"
638:                                          <RIGHTBRACKET                  >
639: <ACTIONCALL                        > ::=  "#"
640:                                           NUMERIC LITERAL
641: <ACTIONCALL                        > ::=  "@"
642:                                          #S
643:                                          <SEMANTICROUTINECALL           >
644: <ACTIONCALL                        > ::=  "@"
```

```
                                        #Q
                                        <SEMANTICROUTINECALL                >
<ACTIONCALL                     > ::=  "@"
                                        #S
<ACTIONCALL                     > ::=  "@"
                                        #Q
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 134                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 135                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 136                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 137                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 138                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 139                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 140                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 141                          >
<BITTEST                        > ::=  "@"
                                        #T
                                        <DUMMY 142                          >
<BITACTION                      > ::=  "@"
                                        #S
                                        <DUMMY 143                          >
<BITACTION                      > ::=  "@"
                                        #S
                                        <DUMMY 144                          >
<SEMANTICROUTINECALL            > ::=  ANY SYMBOL    8
                                        ";"
                                        <CODEDECLARATION                    >
<SEMANTICROUTINECALL            > ::=  ANY SYMBOL    8
<SEMANTICROUTINECALL            > ::= <SEMANTICROUTI DUMMY 15               >
                                        ";"
                                        <CODEDECLARATION                    >
<SEMANTICROUTINECALL            > ::= <SEMANTICROUTI DUMMY 15               >
<SEMANTICROUTINECALL            > ::= <CODEDECLARATION                      >
<SEMANTICROUTINECALL            > ::=  ANY SYMBOL    8
                                        ";"
<SEMANTICROUTINECALL            > ::= <SEMANTICROUTI DUMMY 15               >
                                        ";"
<SEMANTICROUTI DUMMY 15         > ::= <SEMANTICROUTI DUMMY 15               >
                                        ","
                                        NUMERIC LITERAL
<SEMANTICROUTI DUMMY 15         > ::=  "("
                                        NUMERIC LITERAL
<SEMANTICROUTI DUMMY 15         > ::=  ANY SYMBOL    8
                                        NUMERIC LITERAL
```

```
704: <CODEDECLARATION                > ::= <CODE DUMMY 16                >
705: <CODEDECLARATION                > ::= #BEGIN
706:                                       <CODE DUMMY 16                >
707:                                       #END
708: <CODEDECLARATION                > ::= #BEGIN
709:                                       #END
710: <CODE DUMMY 16                  > ::= <CODE DUMMY 16                >
711:                                       <DUMMY 145                    >
712: <CODE DUMMY 16                  > ::= <CODE DUMMY 16                >
713:                                       <CODEDECLARATION              >
714: <CODE DUMMY 16                  > ::=  ANY SYMBOL     9
715: <CODE DUMMY 16                  > ::= <CODEDECLARATION              >
716: <CODE DUMMY 16                  > ::= <CODE DUMMY 16                >
717: <NULLSYMBOL DUMMY 17            > ::= #COMMENT
718:                                        "."
719: <NULLSYMBOL DUMMY 17            > ::= #C
720:                                        "."
721: <NULLSYMBOL DUMMY 17            > ::= <NULLSYMBOL DUMMY 18          >
722:                                        "."
723: <NULLSYMBOL DUMMY 18            > ::= <NULLSYMBOL DUMMY 18          >
724:                                        ANY SYMBOL    10
725: <NULLSYMBOL DUMMY 18            > ::= #COMMENT
726:                                        ANY SYMBOL    10
727: <NULLSYMBOL DUMMY 18            > ::= #C
728:                                        ANY SYMBOL    10
729: <NONTERMINAL DUMMY 19           > ::= <NONTERMINAL DUMMY 19         >
730:                                       <DUMMY 78                     >
731: <NONTERMINAL DUMMY 19           > ::=  ANY SYMBOL    12
732:                                       <DUMMY 146                    >
733: <NONTERMINAL DUMMY 19           > ::= <LEFTBRACKET                  >
734:                                       <DUMMY 77                     >
735: <PRINTOPTION                    > ::= #TABLESF
736:                                       #SIZES
737: <PRINTOPTION                    > ::= #CHARACTERS
738: <PRINTOPTION                    > ::= #TERMINALS
739: <PRINTOPTION                    > ::= #TERMINALS
740:                                       #ALPHABETICALLY
741: <PRINTOPTION                    > ::= #TRMCHR
742: <PRINTOPTION                    > ::= #TERMINALS
743:                                       #NUMERICALLY
744: <PRINTOPTION                    > ::= #TRMALF
745: <PRINTOPTION                    > ::= #TRMNUM
746: <PRINTOPTION                    > ::= #NONTERMINALS
747: <PRINTOPTION                    > ::= #NONTERMINALS
748:                                       #ALPHABETICALLY
749: <PRINTOPTION                    > ::= #NTALF
750: <PRINTOPTION                    > ::= #NONTERMINALS
751:                                       #NUMERICALLY
752: <PRINTOPTION                    > ::= #NTNUM
753: <PRINTOPTION                    > ::= #SYNTAX
754: <PRINTOPTION                    > ::= #INPUT
755: <PRINTOPTION                    > ::= #CROSS
756:                                       #REFERENCE
757: <PRINTOPTION                    > ::= #XREF
758: <PRINTOPTION                    > ::= #INDEX
759: <PRINTOPTION                    > ::= #PATTERNS
760: <PRINTOPTION                    > ::= #ACTIONS
761: <PRINTOPTION                    > ::= #ACTALF
762: <PRINTOPTION                    > ::= #ACTIONS
```

```
763:                                        #ALPHABETICALLY
764: <PRINTOPTION                    > ::= #ACTNUM
765: <PRINTOPTION                    > ::= #ACTIONS
766:                                        #NUMERICALLY
767: <PRINTOPTION                    > ::= #PROTAB
768: <PRINTOPTION                    > ::= #FLOYD
769: <PRINTOPTION                    > ::= #COMBINED
770:                                        #GROUPS
771: <PRINTOPTION                    > ::= #PARSER
772: <PRINTOPTION                    > ::= #STANDARD
773: <PRINTOPTION                    > ::= #DEBUGGING
774: <PRINTOPTION                    > ::= #DEBUGN
775: <PRINTOPTION                    > ::= #EVERYTHING
776: <PRINTOPTION                    > ::= #NOTHING
777: <CONTROLSTATEMENT DUMMY 20      > ::= <CONTROLSTATEMENT DUMMY 20      >
778:                                        ","
779:                                        <PRINTOPTION                    >
780: <CONTROLSTATEMENT DUMMY 20      > ::= #PRINT
781:                                        ";"
782:                                        <PRINTOPTION                    >
783: <PROGRAMNAME                     > ::= #TWST
784: <PROGRAMNAME                     > ::= #COMPILER
785: <PROGRAMNAME                     > ::= #ALGOL
786: <PROGRAMNAME                     > ::= #ISL
787: <PROGRAMNAME                     > ::= #BNF2FPL
788: <PROGRAMNAME                     > ::= #FPL2PAR
789: <PROGRAMNAME                     > ::= #PAR2ALG
790: <CONTROLSTATEMENT DUMMY 21      > ::= <CONTROLSTATEMENT DUMMY 21      >
791:                                        ","
792:                                        ANY SYMBOL   13
793: <CONTROLSTATEMENT DUMMY 21      > ::= #SPECIAL
794:                                        #SYMBOLS
795:                                        ";"
796:                                        ANY SYMBOL   13
797: <DUMMY 73                        > ::= ";"
798: <DUMMY 74                        > ::= ";"
799:                                        <PRODUCTION DUMMY 4             >
800: <DUMMY 75                        > ::= ";"
801:                                        <CODEDECLARATION                >
802: <DUMMY 76                        > ::= ANY SYMBOL   12
803:                                        <RIGHTBRACKET                   >
804: <DUMMY 77                        > ::= ANY SYMBOL   12
805:                                        <DUMMY 147                      >
806: <DUMMY 78                        > ::= ANY SYMBOL   11
807: <DUMMY 79                        > ::= <NONTERMINAL                    >
808:                                        <ARROW                          >
809:                                        <DEFINITION                     >
810: <DUMMY 80                        > ::= <NONTERMINAL                    >
811:                                        <ARROW                          >
812:                                        <DEFINITION                     >
813: <DUMMY 81                        > ::= <NONTERMINAL                    >
814:                                        #WHICH
815:                                        <ARROW                          >
816:                                        <DEFINITION                     >
817: <DUMMY 82                        > ::= #FOLLOWED
818:                                        #BY
819:                                        <DUMMY 148                      >
820: <DUMMY 83                        > ::= #FOLLOWED
821:                                        #BY
```

```
                                        <SYMBOL                          >
<DUMMY 84                          > ::= #FOLLOWED
                                        #BY
                                        <DUMMY 150                       >
<DUMMY 85                          > ::= #FOLLOWED
                                        #BY
                                        <SYMBOL                          >
<DUMMY 86                          > ::= #FOLLOWED
                                        #BY
                                        <DUMMY 152                       >
<DUMMY 87                          > ::= #FOLLOWED
                                        #BY
                                        <SYMBOL                          >
<DUMMY 88                          > ::= #EQUAL
                                        #SIGN
<DUMMY 89                          > ::= #EQUAL
                                        #SIGNS
<DUMMY 90                          > ::= <NESTEDPRODUCTION DUMMY 8        >
<DUMMY 91                          > ::= <NESTEDPRODUCTION DUMMY 8        >
                                        <KLEENESTAR                      >
<DUMMY 92                          > ::= #PARENTHESES
<DUMMY 93                          > ::= #SQUARE
                                        #BRACKETS
<DUMMY 94                          > ::= #ANGLE
                                        #BRACKETS
<DUMMY 95                          > ::= <PRODUCTION DUMMY 3              >
                                        <DUMMY 96                        >
<DUMMY 96                          > ::= <NONTERMINAL                     >
                                        <ARROW                           >
                                        <DEFINITION                      >
<DUMMY 97                          > ::= <PRODUCTION DUMMY 3              >
                                        <DUMMY 98                        >
<DUMMY 98                          > ::= <NONTERMINAL                     >
                                        #WHICH
                                        <ARROW                           >
                                        <DEFINITION                      >
<DUMMY 99                          > ::= <NONTERMINAL                     >
                                        <ARROW                           >
                                        <DEFINITION                      >
<DUMMY 100                         > ::= <NONTERMINAL                     >
                                        #WHICH
                                        <ARROW                           >
                                        <DEFINITION                      >
<DUMMY 101                         > ::= <SEEDEDLIST                      >
<DUMMY 102                         > ::= <PRODUCTION DUMMY 3              >
                                        <DUMMY 104                       >
<DUMMY 103                         > ::= <SYMBOL                          >
<DUMMY 104                         > ::= <SIMPLESYNTACTICSYMBOL           >
<DUMMY 105                         > ::= <SYMBOL                          >
                                        <DUMMY 84                        >
<DUMMY 106                         > ::= <SYMBOL                          >
                                        <SYMBOL                          >
<DUMMY 107                         > ::= <SYMBOL                          >
                                        <DUMMY 85                        >
<DUMMY 108                         > ::= <PRODUCTION DUMMY 3              >
                                        <DUMMY 101                       >
<DUMMY 109                         > ::= <SEEDEDLIST                      >
<DUMMY 110                         > ::= <PRODUCTION DUMMY 3              >
                                        <DUMMY 111                       >
```

```
881: <DUMMY 111                  > ::= <SIMPLESYNTACTICSYMBOL          >
882: <DUMMY 112                  > ::= <PRODUCTION DUMMY 3             >
883:                                   <DUMMY 109                      >
884: <DUMMY 113                  > ::= <SEEDEDLIST                     >
885: <DUMMY 114                  > ::= <SEEDEDLIST                     >
886: <DUMMY 115                  > ::= #DIGIT
887: <DUMMY 116                  > ::= #DIGITS
888: <DUMMY 117                  > ::= #LETTER
889: <DUMMY 118                  > ::= #LETTERS
890: <DUMMY 119                  > ::= #SPECIAL
891:                                   #CHARACTER
892: <DUMMY 120                  > ::= #SPECIAL
893:                                   #CHARACTERS
894: <DUMMY 121                  > ::= #CHARACTER
895: <DUMMY 122                  > ::= #CHARACTERS
896: <DUMMY 123                  > ::= #RELATIONAL
897:                                   #OPERATOR
898: <DUMMY 124                  > ::= #RELATIONAL
899:                                   #OPERATORS
900: <DUMMY 125                  > ::= #NON
901:                                   #CHARACTER
902: <DUMMY 126                  > ::= #NON
903:                                   #CHARACTERS
904: <DUMMY 127                  > ::= #NONCHARACTER
905: <DUMMY 128                  > ::= #NONCHARACTERS
906: <DUMMY 129                  > ::= #TERMINAL
907: <DUMMY 130                  > ::= #TERMINALS
908: <DUMMY 131                  > ::=  "("
909:                                   <DUMMY 132                      >
910: <DUMMY 132                  > ::= <TERMINAL                       >
911: <DUMMY 133                  > ::= #OR
912:                                   <TERMINAL                       >
913: <DUMMY 134                  > ::= #BIT
914:                                    ANY SYMBOL     5
915:                                   #ON
916:                                   #SET
917: <DUMMY 135                  > ::= #BIT
918:                                    ANY SYMBOL     5
919:                                   #ON
920:                                   #RESET
921: <DUMMY 136                  > ::= #BIT
922:                                    ANY SYMBOL     5
923:                                   #ON
924: <DUMMY 137                  > ::= #BIT
925:                                    ANY SYMBOL     5
926:                                   #OFF
927:                                   #SET
928: <DUMMY 138                  > ::= #BIT
929:                                    ANY SYMBOL     5
930:                                   #OFF
931:                                   #RESET
932: <DUMMY 139                  > ::= #BIT
933:                                    ANY SYMBOL     5
934:                                   #OFF
935: <DUMMY 140                  > ::= #BIT
936:                                    ANY SYMBOL     5
937:                                   #SET
938: <DUMMY 141                  > ::= #BIT
939:                                    ANY SYMBOL     5
```

```
                                        #RESET
<DUMMY 142                        > ::= #BIT
                                         ANY SYMBOL    5
<DUMMY 143                        > ::= #SET
                                        #BIT
                                         ANY SYMBOL    5
<DUMMY 144                        > ::= #RESET
                                        #BIT
                                         ANY SYMBOL    5
<DUMMY 145                        > ::=  ANY SYMBOL    9
<DUMMY 146                        > ::=  ANY SYMBOL   11
<DUMMY 147                        > ::=  ANY SYMBOL   11
<DUMMY 148                        > ::= <PRODUCTION DUMMY 3          >
                                        <DUMMY 149                   >
<DUMMY 149                        > ::= <SYMBOL                      >
<DUMMY 150                        > ::= <PRODUCTION DUMMY 3          >
                                        <DUMMY 151                   >
<DUMMY 151                        > ::= <SYMBOL                      >
<DUMMY 152                        > ::= <PRODUCTION DUMMY 3          >
                                        <DUMMY 153                   >
<DUMMY 153                        > ::= <SYMBOL                      >
```

LIST OF REFERENCES

[1] Naur, P., "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, 6 (Jan., 1963), pp. 1-17.

[2] Perstein, M.H., "Some Techniques for Describing Programming Languages", SP-2916/000/01, System Development Corporation, Santa Monica, California (1968).

[3] Beals, A.J., "The Generation of a Deterministic Parsing Algorithm", Report No. 304, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois (1969).

[4] Trout, H.R.G., "A BNF Like Language for the Description of Syntax Directed Compilers", Report No. 300, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois (1969).

[5] Iverson, K.E., "A Method of Syntax Specification", Comm ACM, 7 (Oct., 1964), pp. 588-589.

[6] Floyd, R.W., "A Descriptive Language for Symbol Manipulation", JACM, 8 (Oct., 1961), pp. 579-584.

[7] Floyd, R.W., "Syntactic Analysis and Operator Precedence", JACM, 10 (July, 1963), pp. 316-333.

[8] Leavenworth, B.M., "FORTRAN IV as a Syntax Language", Comm ACM, 7 (Feb., 1964), pp. 72-80.

[9] Balzer, R.M., and Farber, D.J., "APAREL-A Parse-Request Language", Comm ACM, 12 (Nov., 1969), pp. 624-631.

[10] Machado, N.C., "ISL-A Semantics Language for a Translator Writing System", Report No. 367, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, (1969).

[11] Brooker, R.A. et al, "The Compiler-Compiler," Annual Review in Automatic Programming, 3 (1963) p. 229.

[12] Floyd, R.W., "The Syntax of Programming Languages, A Survey", IEEE Trans., EC13, 4 (Aug., 1964), pp. 346-353.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY *(Corporate author)*

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3 REPORT TITLE

TWINKLE--A SYNTAX LANGUAGE FOR A TRANSLATOR WRITING SYSTEM

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Research Report

5. AUTHOR(S) *(First name, middle initial, last name)*

Robert Leroy Mercer

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 15, 1970	102	12

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
USAF 30(602)4144	
b. PROJECT NO. 46-26-15-305	DCS Report No. 396
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	ILLIAC IV Document No. 218

10 DISTRIBUTION STATEMENT

Qualified requesters may obtain copies from DCS.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Rome Air Development Center Griffiss Air Force Base Rome, New York 13440

13 ABSTRACT

TWINKLE is a language designed to aid in the syntactic specification of programming languages. In addition to the constructs available in BNF, TWINKLE provides for easy specification of lists and other frequently used linguistic structures. By providing a large number of alternatives for its various constructs, TWINKLE allows the language designer to specify a language in terms that approach natural language.

The implementation of a compiler for TWINKLE is described. This compiler is the first phase of the ILLIAC IV Translator Writing System.

DD FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Compiler-Compiler						
Language Translation and Linguistics						
Programming						
Compilers and Generators						
Procedure and Problem Oriented Languages						

MAY 10 1973

BOUND BY
HM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.391-396(1970
Digital computer internal report /
3 0112 088399149